

04. 探寻插件的原理与深究和Spring的集成

课程目标

1. 掌握MyBatis插件的使用和工作原理
2. 掌握自定义插件的编写方法
3. 掌握Spring集成MyBatis的原理

内容定位

- 1、对MyBatis插件工作原理不清楚的同学
- 2、对于Spring集成MyBatis原理不了解的同学

一、MyBatis插件

插件是一种常见的扩展方式，大多数开源框架也都支持用户通过添加自定义插件的方式来扩展或者改变原有的功能，MyBatis中也提供的有插件，虽然叫插件，但是实际上是通过拦截器(Interceptor)实现的，在MyBatis的插件模块中涉及到责任链模式和DK动态代理，这两种设计模式的技术知识也是大家要提前掌握的。

1. 自定义插件

首先我们来看下一个自定义的插件我们要如何来实现。

1.1 创建Interceptor实现类

我们创建的拦截器必须要实现Interceptor接口，Interceptor接口的定义为

```
public interface Interceptor {  
  
    // 执行拦截逻辑的方法  
    Object intercept(Invocation invocation) throws Throwable;  
  
    // 决定是否触发 intercept()方法  
    default Object plugin(Object target) {  
        return Plugin.wrap(target, this);  
    }  
  
    // 根据配置 初始化 Interceptor 对象  
    default void setProperties(Properties properties) {  
        // NOP  
    }  
}
```

在MyBatis中Interceptor允许拦截的内容是

- Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

- ParameterHandler (getParameterObject, setParameters)
- ResultSetHandler (handleResultSets, handleOutputParameters)
- StatementHandler (prepare, parameterize, batch, update, query)

我们创建一个拦截Executor中的query和close的方法

```

package com.gupaoedu.interceptor;

import org.apache.ibatis.executor.Executor;
import org.apache.ibatis.mapping.MappedStatement;
import org.apache.ibatis.plugin.*;
import org.apache.ibatis.session.ResultHandler;
import org.apache.ibatis.session.RowBounds;

import java.util.Properties;

/**
 * 自定义的拦截器
 * @Signature 注解就可以表示一个方法签名， 唯一确定一个方法
 */
@Intercepts({
    @Signature(
        type = Executor.class // 需要拦截的类型
        ,method = "query" // 需要拦截的方法
        // args 中指定 被拦截方法的 参数列表
        ,args={MappedStatement.class, Object.class, RowBounds.class,
        ResultHandler.class}
    ),
    @Signature(
        type = Executor.class
        ,method = "close"
        ,args = {boolean.class}
    )
})
public class FirstInterceptor implements Interceptor {

    private int testProp;

    /**
     * 执行拦截逻辑的方法
     * @param invocation
     * @return
     * @throws Throwable
     */
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        System.out.println("FirstInterceptor 拦截之前 ....");
        Object obj = invocation.proceed();
        System.out.println("FirstInterceptor 拦截之后 ....");
        return obj;
    }

    /**
     * 决定是否触发 intercept方法
     * @param target
     * @return
     */
}

```

```

/*
@Override
public Object plugin(Object target) {
    return Plugin.wrap(target, this);
}

@Override
public void setProperties(Properties properties) {
    System.out.println("---->"+properties.get("testProp"));
}

public int getTestProp() {
    return testProp;
}

public void setTestProp(int testProp) {
    this.testProp = testProp;
}
}

```

1.2 配置拦截器

创建好自定义的拦截器后，我们需要在全局配置文件中添加自定义插件的注册

```

<plugins>
    <plugin interceptor="com.gupaoedu.interceptor.FirstInterceptor">
        <property name="testProp" value="1000"/>
    </plugin>
</plugins>

```

1.3 运行程序

然后我们执行对应的查询操作。

```

@Test
public void test1() throws Exception{
    // 1.获取配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2.加载解析配置文件并获取SqlSessionFactory对象
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3.根据SqlSessionFactory对象获取SqlSession对象
    SqlSession sqlSession = factory.openSession();
    // 4.通过SqlSession中提供的 API方法来操作数据库
    List<User> list =
    sqlSession.selectList("com.gupaoedu.mapper.UserMapper.select userList");
    for (User user : list) {
        System.out.println(user);
    }
    // 5.关闭会话
    sqlSession.close();
}

```

在输出语句中可以看到我们的日志信息

```
public void test1() throws Exception{
    // 1. 获取配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2. 加载解析配置文件并获取SqlSessionFactory对象
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3. 根据SqlSessionFactory对象获取SqlSession对象
}
---->1000
PooledDataSource forcefully closed/removed all connections.
FirstInterceptor 拦截之前 ....
Cache Hit Ratio [com.gupaoedu.mapper.UserMapper]: 1.0
FirstInterceptor 拦截之后 ....
User(id=1, userName=zhangsan, realName=张三, password=123456, age=18, dId=null)
User(id=2, userName=lisi, realName=李四, password=1111, age=19, dId=null)
User(id=3, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=4, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=5, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=6, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=7, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=8, userName=aaa, realName=bbb, password=null, age=null, dId=null)
User(id=9, userName=aaa, realName=bbb, password=null, age=null, dId=null)
User(id=10, userName=aaa, realName=bbb, password=null, age=null, dId=null)
User(id=11, userName=aaa, realName=bbb, password=null, age=null, dId=null)
```

拦截的query方法和close方法的源码位置在如下：

```
@Override
public <T> List<T> selectList(String statement, Object parameter, RowBounds rowBounds) {
    MappedStatement ms = configuration.getMappedStatement(statement);
    // 如果缓存已启用，则使用缓存
    if (cacheEnabled == true) {
        return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
    }
    catch (Exception e) {
        throw new ExecutorException("Statement [" + statement + "] was never mapped. " +
            "Object [" + parameter + "] with RowBounds [" + rowBounds + "] was passed to query(" +
            "java.util.List<?>) without ResultHandler.", e);
    }
    finally {
        ErrorContext.instance().reset();
    }
}
@Override
public void select(String statement, Object parameter, ResultHandler handler) {
```

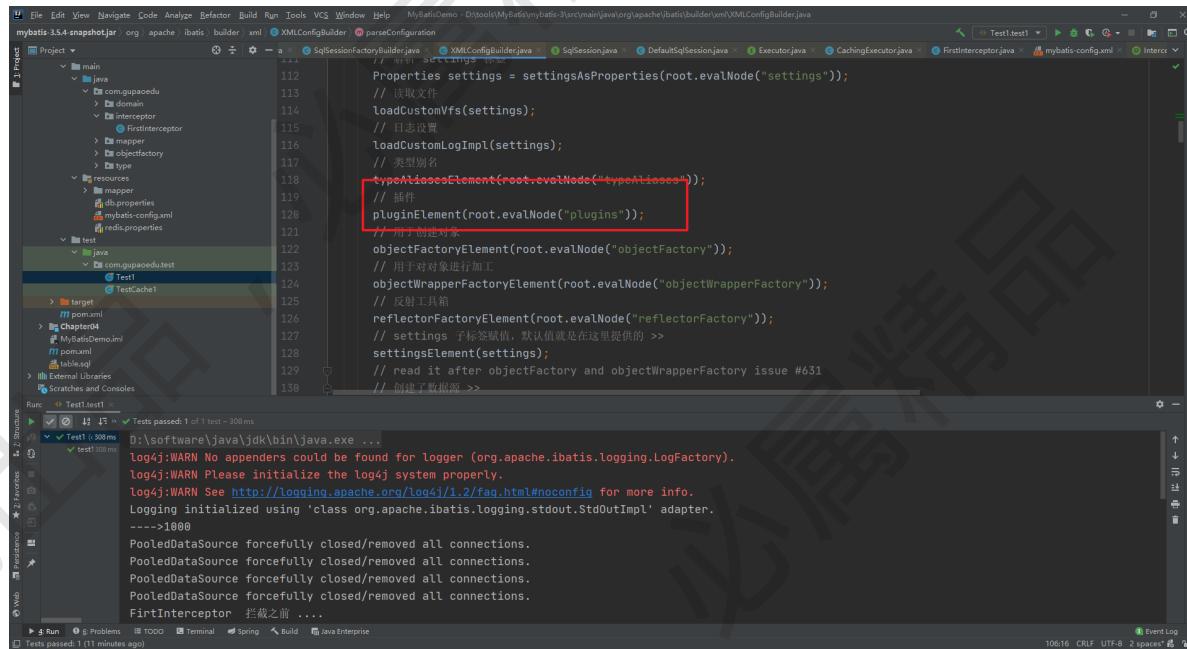
```
@Override
public void close() {
    try {
        executor.close(isCommitOrRollbackRequired(force: false));
    }
    catch (Exception e) {
        throw new ExecutorException("Statement [" + statement + "] was never mapped. " +
            "Object [" + parameter + "] with RowBounds [" + rowBounds + "] was passed to query(" +
            "java.util.List<?>) without ResultHandler.", e);
    }
    finally {
        ErrorContext.instance().reset();
    }
}
private void closeCursors() {
    if (cursorList != null && !cursorList.isEmpty()) {
        for (Cursor cursor : cursorList) {
```

2. 插件实现原理

自定义插件的步骤还是比较简单的，接下来我们分析下插件的实现原理是怎么回事。

2.1 初始化操作

首先我们来看下在全局配置文件加载解析的时候做了什么操作。



进入方法内部可以看到具体的解析操作

```
private void pluginElement(XNode parent) throws Exception {
    if (parent != null) {
        for (XNode child : parent.getChildren()) {
            // 获取<plugin> 节点的 interceptor 属性的值
            String interceptor = child.getStringAttribute("interceptor");
            // 获取<plugin> 下的所有的properties子节点
            Properties properties = child.getChildrenAsProperties();
            // 获取 Interceptor 对象
            Interceptor interceptorInstance = (Interceptor)
                resolveClass(interceptor).getDeclaredConstructor().newInstance();
            // 设置 interceptor的 属性
            interceptorInstance.setProperties(properties);
            // Configuration中记录 Interceptor
            configuration.addInterceptor(interceptorInstance);
        }
    }
}
```

该方法用来解析全局配置文件中的plugins标签，然后对应的创建Interceptor对象，并且封装对应的属性信息。最后调用了Configuration对象中的方法。

```
configuration.addInterceptor(interceptorInstance)
```

```
public void addInterceptor(Interceptor interceptor) {  
    interceptorChain.addInterceptor(interceptor);  
}
```

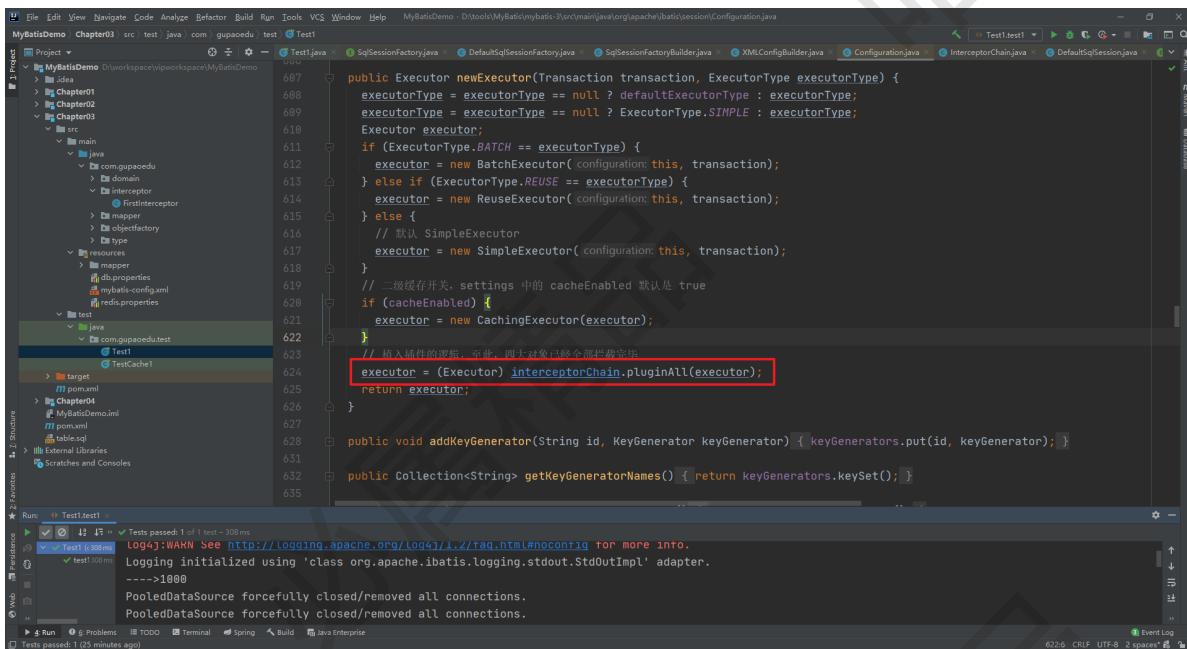
通过这个代码我们发现我们自定义的拦截器最终是保存在了InterceptorChain这个对象中。而InterceptorChain的定义为

```
public class InterceptorChain {  
  
    // 保存所有的 Interceptor 也就我所有的插件是保存在 Interceptors 这个List集合中的  
    private final List<Interceptor> interceptors = new ArrayList<>();  
  
    //  
    public Object pluginAll(Object target) {  
        for (Interceptor interceptor : interceptors) { // 获取拦截器链中的所有拦截器  
            target = interceptor.plugin(target); // 创建对应的拦截器的代理对象  
        }  
        return target;  
    }  
  
    public void addInterceptor(Interceptor interceptor) {  
        interceptors.add(interceptor);  
    }  
  
    public List<Interceptor> getInterceptors() {  
        return Collections.unmodifiableList(interceptors);  
    }  
}
```

2.2 如何创建代理对象

在解析的时候创建了对应的Interceptor对象，并保存在了InterceptorChain中，那么这个拦截器是如何和对应的目标对象进行关联的呢？首先拦截器可以拦截的对象是Executor,ParameterHandler,ResultSetHandler,StatementHandler.那么我们来看下这四个对象在创建的时候又什么要注意的

2.2.1 Executor



我们可以看到Executor在装饰完二级缓存后会通过pluginAll来创建Executor的代理对象。

```
public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) { // 获取拦截器链中的所有拦截器
        target = interceptor.plugin(target); // 创建对应的拦截器的代理对象
    }
    return target;
}
```

进入plugin方法中，我们会进入到

```
// 决定是否触发 intercept()方法
default Object plugin(Object target) {
    return Plugin.wrap(target, this);
}
```

然后进入到MyBatis给我们提供的Plugin工具类的实现 wrap方法中。

```
/**
 * 创建目标对象的代理对象
 * 目标对象 Executor ParameterHandler ResultSetHandler StatementHandler
 * @param target 目标对象
 * @param interceptor 拦截器
 * @return
 */
public static Object wrap(Object target, Interceptor interceptor) {
    // 获取用户自定义 Interceptor中@Signature注解的信息
    // getSignatureMap 负责处理@Signature 注解
    Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
    // 获取目标类型
    Class<?> type = target.getClass();
    // 获取目标类型 实现的所有的接口
    Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
    // 如果目标类型有实现的接口 就创建代理对象
    if (interfaces.length > 0) {
        return Proxy.newProxyInstance(
            type.getClassLoader(),
            interfaces,
            interceptor);
    }
}
```

```
        interfaces,
        new Plugin(target, interceptor, signatureMap));
    }
    // 否则原封不动的返回目标对象
    return target;
}
```

Plugin中的各个方法的作用

```
public class Plugin implements InvocationHandler {

    private final Object target; // 目标对象
    private final Interceptor interceptor; // 拦截器
    private final Map<Class<?>, Set<Method>> signatureMap; // 记录 @Signature 注解的信息

    private Plugin(Object target, Interceptor interceptor, Map<Class<?>,
Set<Method>> signatureMap) {
        this.target = target;
        this.interceptor = interceptor;
        this.signatureMap = signatureMap;
    }

    /**
     * 创建目标对象的代理对象
     * 目标对象 Executor ParameterHandler ResultSetHandler StatementHandler
     * @param target 目标对象
     * @param interceptor 拦截器
     * @return
     */
    public static Object wrap(Object target, Interceptor interceptor) {
        // 获取用户自定义 Interceptor中@Signature注解的信息
        // getSignatureMap 负责处理@Signature 注解
        Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
        // 获取目标类型
        Class<?> type = target.getClass();
        // 获取目标类型 实现的所有的接口
        Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
        // 如果目标类型有实现的接口 就创建代理对象
        if (interfaces.length > 0) {
            return Proxy.newProxyInstance(
                type.getClassLoader(),
                interfaces,
                new Plugin(target, interceptor, signatureMap));
        }
        // 否则原封不动的返回目标对象
        return target;
    }

    /**
     * 代理对象方法被调用时执行的代码
     * @param proxy
     * @param method
     * @param args
     * @return
     * @throws Throwable
     */
}
```

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        // 获取当前方法所在类或接口中，可被当前Interceptor拦截的方法
        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
        if (methods != null && methods.contains(method)) {
            // 当前调用的方法需要被拦截 执行拦截操作
            return interceptor.intercept(new Invocation(target, method, args));
        }
        // 不需要拦截 则调用 目标对象中的方法
        return method.invoke(target, args);
    } catch (Exception e) {
        throw ExceptionUtil.unwrapThrowable(e);
    }
}

private static Map<Class<?>, Set<Method>> getSignatureMap(Interceptor
interceptor) {
    Intercepts interceptsAnnotation =
interceptor.getClass().getAnnotation(Intercepts.class);
    // issue #251
    if (interceptsAnnotation == null) {
        throw new PluginException("No @Intercepts annotation was found in
interceptor " + interceptor.getClass().getName());
    }
    Signature[] sigs = interceptsAnnotation.value();
    Map<Class<?>, Set<Method>> signatureMap = new HashMap<>();
    for (Signature sig : sigs) {
        Set<Method> methods = signatureMap.computeIfAbsent(sig.type(), k -> new
HashSet<>());
        try {
            Method method = sig.type().getMethod(sig.method(), sig.args());
            methods.add(method);
        } catch (NoSuchMethodException e) {
            throw new PluginException("Could not find method on " + sig.type() +
named " + sig.method() + ". Cause: " + e, e);
        }
    }
    return signatureMap;
}

private static Class<?>[] getAllInterfaces(Class<?> type, Map<Class<?>,
Set<Method>> signatureMap) {
    Set<Class<?>> interfaces = new HashSet<>();
    while (type != null) {
        for (Class<?> c : type.getInterfaces()) {
            if (signatureMap.containsKey(c)) {
                interfaces.add(c);
            }
        }
        type = type.getSuperclass();
    }
    return interfaces.toArray(new Class<?>[interfaces.size()]);
}
```

2.2.2 StatementHandler

在获取StatementHandler的方法中

```
@Override  
public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {  
    Statement stmt = null;  
    try {  
        Configuration configuration = ms.getConfiguration();  
        // 注意，已经来到SQL处理的关键对象 StatementHandler >>  
        StatementHandler handler = configuration.newStatementHandler(wrapper, ms,  
parameter, rowBounds, resultHandler, boundSql);  
        // 获取一个 Statement对象  
        stmt = prepareStatement(handler, ms.createStatementLog());  
        // 执行查询  
        return handler.query(stmt, resultHandler);  
    } finally {  
        // 用完就关闭  
        closeStatement(stmt);  
    }  
}
```

在进入newStatementHandler方法

```
public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {  
    StatementHandler statementHandler = new RoutingStatementHandler(executor,  
mappedStatement, parameterObject, rowBounds, resultHandler, boundSql);  
    // 植入插件逻辑（返回代理对象）  
    statementHandler = (StatementHandler)  
interceptorChain.pluginAll(statementHandler);  
    return statementHandler;  
}
```

可以看到statementHandler的代理对象

2.2.3 ParameterHandler

在上面步骤的RoutingStatementHandler方法中，我们来看看

```
public RoutingStatementHandler(Executor executor, MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {  
    // StatementType 是怎么来的？ 增删改查标签中的 statementType="PREPARED"，默认值  
    PREPARED  
    switch (ms.getStatementType()) {  
        case STATEMENT:
```

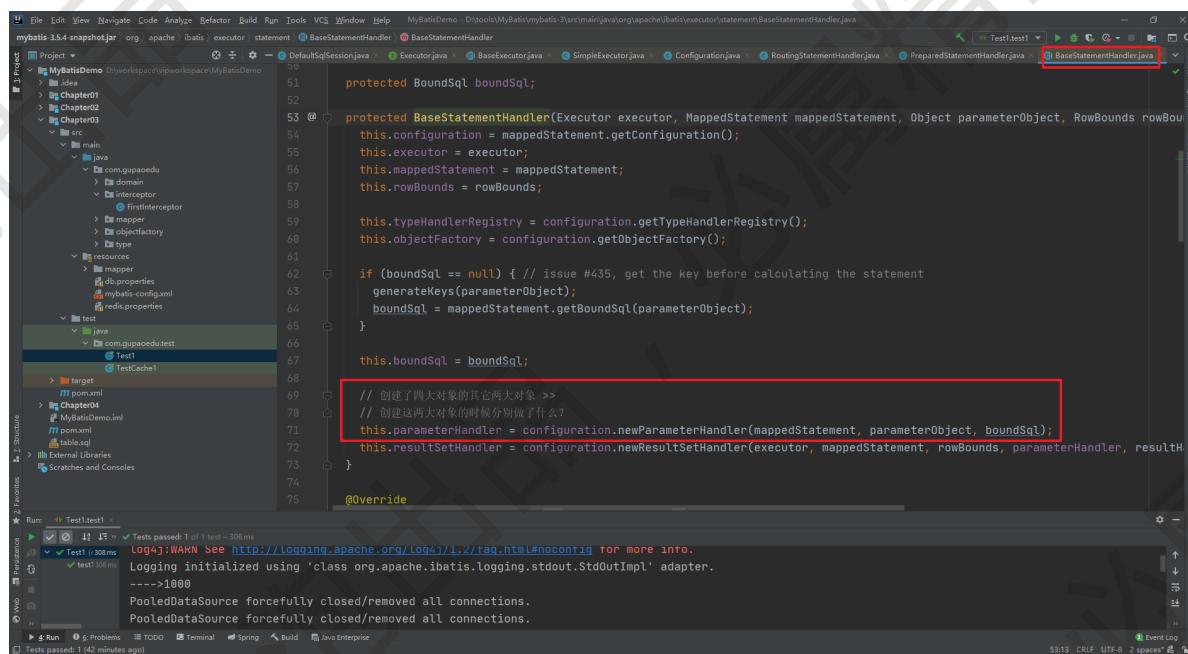
```

        delegate = new SimpleStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundsSql);
        break;
    case PREPARED:
        // 创建 StatementHandler 的时候做了什么? >>
        delegate = new PreparedStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundsSql);
        break;
    case CALLABLE:
        delegate = new CallableStatementHandler(executor, ms, parameter,
rowBounds, resultHandler, boundsSql);
        break;
    default:
        throw new ExecutorException("Unknown statement type: " +
ms.getStatementType());
    }
}

```

}

然后我们随便选择一个分支进入，比如PreparedStatementHandler



在newParameterHandler的步骤我们可以发现代理对象的创建

```

public ParameterHandler newParameterHandler(MappedStatement mappedStatement,
Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
    // 植入插件逻辑（返回代理对象）
    parameterHandler = (ParameterHandler)
interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
}

```

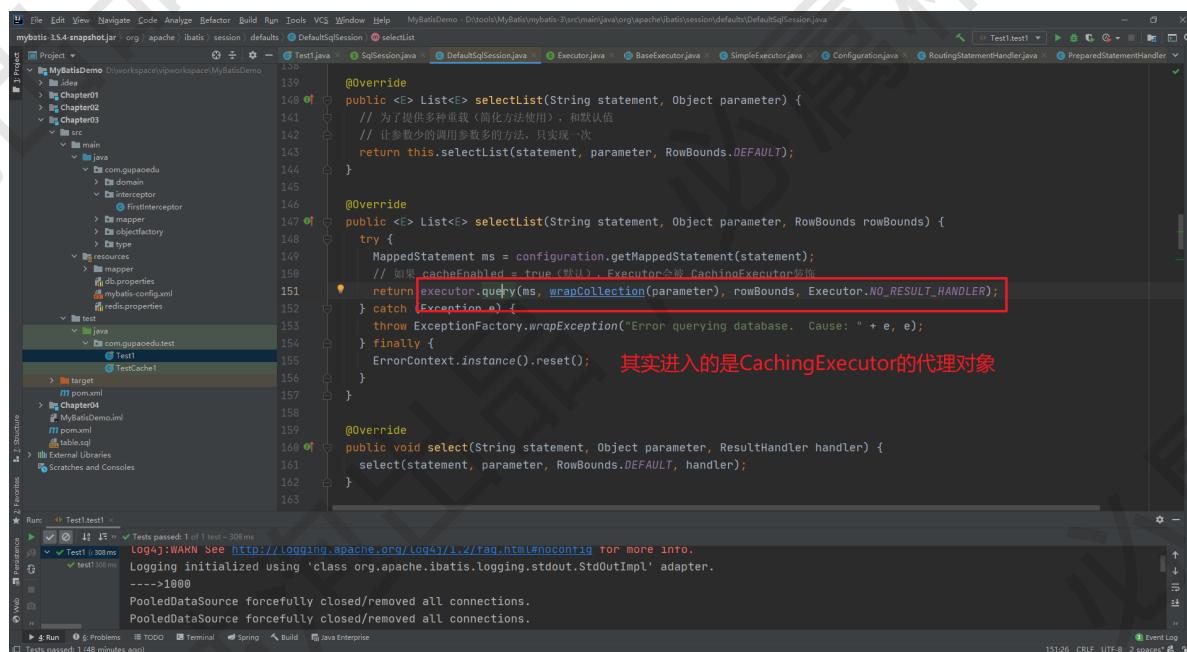
2.2.4 ResultSetHandler

在上面的newResultSetHandler()方法中，也可以看到ResultSetHandler的代理对象

```
public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler parameterHandler, ResultHandler resultHandler, BoundSql boundSql) {  
    ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler, resultHandler, boundSql, rowBounds);  
    // 植入插件逻辑（返回代理对象）  
    resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler);  
    return resultSetHandler;  
}
```

2.3 执行流程

以Executor的query方法为例，当查询请求到来的时候，Executor的代理对象是如何处理拦截请求的呢？我们来看下。当请求到了executor.query方法的时候



然后会执行Plugin的invoke方法

```
/**  
 * 代理对象方法被调用时执行的代码  
 * @param proxy  
 * @param method  
 * @param args  
 * @return  
 * @throws Throwable  
 */  
@Override  
public Object invoke(Object proxy, Method method, Object[] args) throws  
Throwable {  
    try {  
        // 获取当前方法所在类或接口中，可被当前Interceptor拦截的方法  
        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
```

```

if (methods != null && methods.contains(method)) {
    // 当前调用的方法需要被拦截 执行拦截操作
    return interceptor.intercept(new Invocation(target, method, args));
}
// 不需要拦截 则调用 目标对象中的方法
return method.invoke(target, args);
} catch (Exception e) {
    throw ExceptionUtil.unwrapThrowable(e);
}
}

```

然后进入interceptor.intercept 会进入我们自定义的 FirstInterceptor对象中

```

/**
 * 执行拦截逻辑的方法
 * @param invocation
 * @return
 * @throws Throwable
 */
@Override
public Object intercept(Invocation invocation) throws Throwable {
    System.out.println("FirstInterceptor 拦截之前 ....");
    Object obj = invocation.proceed(); // 执行目标方法
    System.out.println("FirstInterceptor 拦截之后 ....");
    return obj;
}

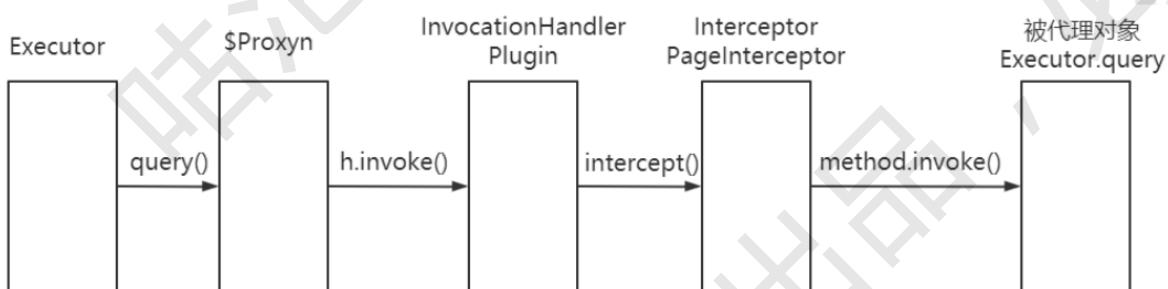
```

这个就是自定义的拦截器执行的完整流程,

2.4 多拦截器

如果我们有多个自定义的拦截器，那么他的执行流程是怎么样的呢？比如我们创建了两个 Interceptor 都是用来拦截 Executor 的query方法，一个是用来执行逻辑A 一个是用来执行逻辑B的。

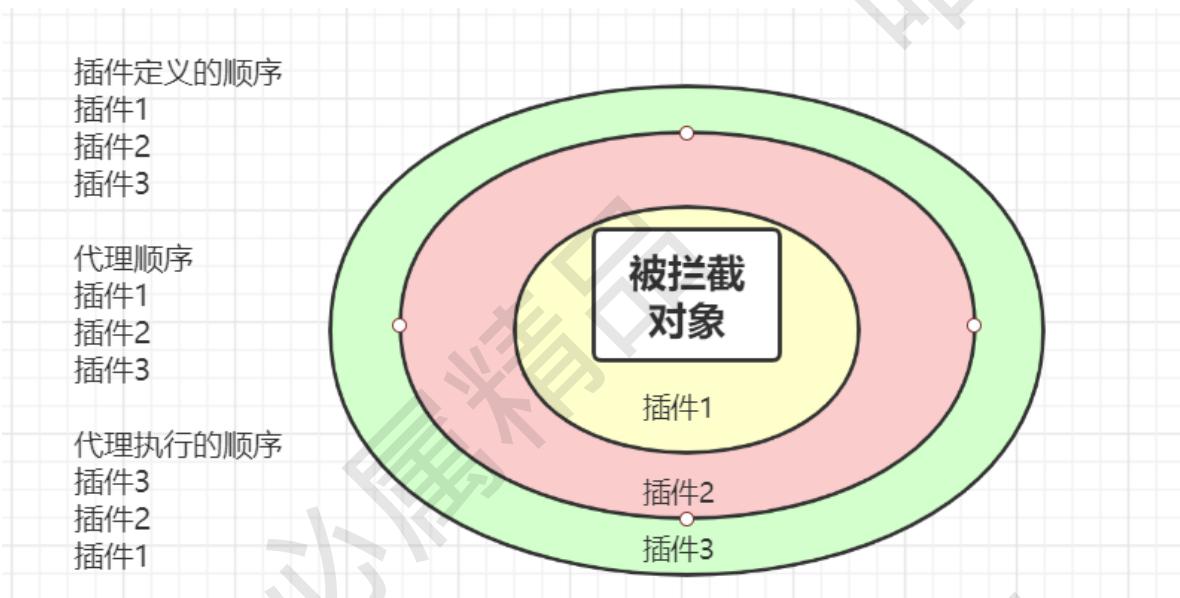
单个拦截器的执行流程



如果说对象被代理了多次，这里会继续调用下一个插件的逻辑，再走一次Plugin的invoke()方法。这里我们需要关注一下有多个插件的时候的运行顺序。

配置的顺序和执行的顺序是相反的。InterceptorChain的List是按照插件从上往下的顺序解析、添加的。

而创建代理的时候也是按照list的顺序代理。执行的时候当然是从最后代理的对象开始。



这个我们可以通过实际的案例来得到验证，最后来总结下Interceptor的相关对象的作用

对象	作用
Interceptor	自定义插件需要实现接口，实现4个方法
InterceptChain	配置的插件解析后会保存在Configuration的InterceptChain中
Plugin	触发管理类，还可以用来创建代理对象
Invocation	对被代理类进行包装，可以调用proceed()调用到被拦截的方法

3. PageHelper分析

Mybatis的插件使用中分页插件PageHelper应该是我们使用到的比较多的插件应用。我们先来回顾下PageHelper的应用

3.1 PageHelper的应用

添加依赖

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>4.1.6</version>
</dependency>
```

在全局配置文件中注册

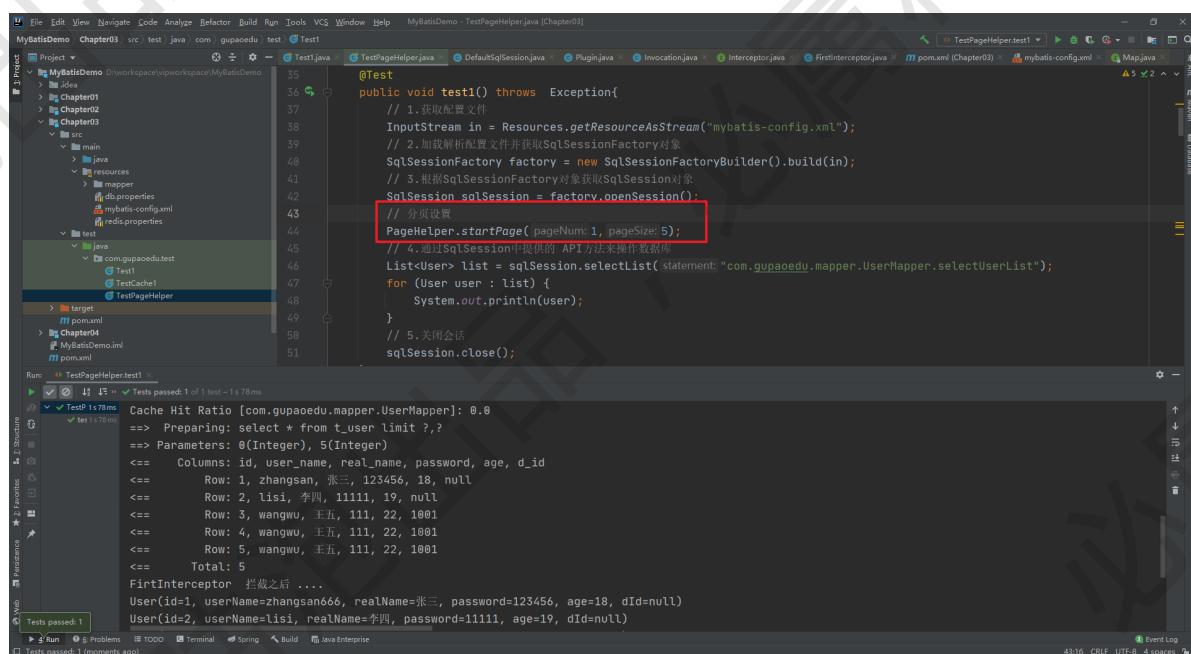
```
<!-- com.github.pagehelper为PageHelper类所在包名 -->
<plugin interceptor="com.github.pagehelper.PageHelper">
    <property name="dialect" value="mysql" />
    <!-- 该参数默认为false -->
    <!-- 设置为true时，会将RowBounds第一个参数offset当成pageNum页码使用 -->
    <!-- 和startPage中的pageNum效果一样 -->
```

```

<property name="offsetAsPageNum" value="true" />
<!-- 该参数默认为false -->
<!-- 设置为true时，使用RowBounds分页会进行count查询 -->
<property name="rowBoundsWithCount" value="true" />
<!-- 设置为true时，如果pageSize=0或者RowBounds.limit = 0就会查询出全部的结果 -->
<!-- （相当于没有执行分页查询，但是返回结果仍然是Page类型） -->
<property name="pageSizeZero" value="true" />
<!-- 3.3.0版本可用 - 分页参数合理化，默认false禁用 -->
<!-- 启用合理化时，如果pageNum<1会查询第一页，如果pageNum>pages会查询最后一页 -->
<!-- 禁用合理化时，如果pageNum<1或pageNum>pages会返回空数据 -->
<property name="reasonable" value="false" />
<!-- 3.5.0版本可用 - 为了支持startPage(Object params)方法 -->
<!-- 增加了一个`params`参数来配置参数映射，用于从Map或ServletRequest中取值 -->
<!-- 可以配置pageNum,pageSize,count,pageSizeZero,reasonable,不配置映射的用默认值
-->
<!-- 不理解该含义的前提下，不要随便复制该配置 -->
<property name="params" value="pageNum=start;pageSize=limit;" />
<!-- always总是返回 PageInfo 类型，check 检查返回类型是否为 PageInfo，none 返回 Page -->
<property name="returnPageInfo" value="check" />
</plugin>

```

然后就是分页查询操作



```

public void test1() throws Exception{
    // 1. 获取配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2. 加载解析配置文件并获取SqlSessionFactory对象
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3. 根据SqlSessionFactory对象获取SqlSession对象
    SqlSession sqlSession = factory.openSession();
    // 分页设置
    PageHelper.startPage( pageNum: 1, pageSize: 5 );
    // 4. 通过SqlSession提供的 API 方法来操作数据库
    List<User> list = sqlSession.selectList( statement: "com.gupaoedu.mapper.UserMapper.selectUserList" );
    for (User user : list) {
        System.out.println(user);
    }
    // 5. 关闭会话
    sqlSession.close();
}

```

Tests passed: 1 of 1 test - 1s 78ms

Cache Hit Ratio [com.gupaoedu.mapper.UserMapper]: 0.0

==> Preparing: select * from t_user limit ?, ?

==> Parameters: 0(Integer), 5(Integer)

<== Columns: id, user_name, real_name, password, age, d_id

<== Row: 1, zhangsan, 张三, 123456, 18, null

<== Row: 2, lisi, 李四, 11111, 19, null

<== Row: 3, wangwu, 王五, 111, 22, 1001

<== Row: 4, wangwu, 王五, 111, 22, 1001

<== Row: 5, wangwu, 王五, 111, 22, 1001

<== Total: 5

FirtInterceptor 拦截之后

User{id=1, userName=zhangsan666, realName=张三, password=123456, age=18, dId=null}

User{id=2, userName=lisi, realName=李四, password=11111, age=19, dId=null}

查看日志我们能够发现查询出来的只有5条记录

```

public void test1() throws Exception{
    // 1. 获取配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2. 加载解析配置文件并获取SqlSessionFactory对象
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3. 根据SqlSessionFactory对象获取SqlSession对象
}

@Test
public void test1() throws Exception{
    // 1. 获取配置文件
    InputStream in = Resources.getResourceAsStream("mybatis-config.xml");
    // 2. 加载解析配置文件并获取SqlSessionFactory对象
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);
    // 3. 根据SqlSessionFactory对象获取SqlSession对象

    // 打开会话
    SqlSession session = factory.openSession();
    // 开启事务
    session.beginTransaction();

    // 使用PageHelper插件
    PageHelper.startPage(1, 5);

    // 执行查询
    List<User> users = session.selectList("com.gupaoedu.mapper.UserMapper.selectAll");
    System.out.println(users);
    session.commit();
}

```

Created connection 1810742349.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6bedbc4d]
==> Preparing: SELECT count(*) FROM t_user
==> Parameters:
<== Columns: count()
<== Row: 13
<== Total: 1
FirstInterceptor 拦截之后
FirstInterceptor 拦截之前
Cache Hit Ratio [com.gupaoedu.mapper.UserMapper]: 0.0
==> Preparing select * from t_user limit ?,?
==> Parameters: ?,5(Integer)
<== Columns: id, user_name, real_name, password, age, d_id
<== Row: 1, zhangsan, 张三, 123456, 18, null
<== Row: 2, lisi, 李四, 11111, 19, null
<== Row: 3, wangwu, 王五, 111, 22, 1001
<== Row: 4, wangwu, 王五, 111, 22, 1001
<== Row: 5, wangwu, 王五, 111, 22, 1001
<== Total: 5
FirstInterceptor 拦截之后
User(id=1, userName=zhangsan66, realName=张三, password=123456, age=18, dId=null)
User(id=2, userName=lisi, realName=李四, password=11111, age=19, dId=null)
User(id=3, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=4, userName=wangwu, realName=王五, password=111, age=22, dId=1001)
User(id=5, userName=wangwu, realName=王五, password=111, age=22, dId=1001)

通过MyBatis的分页插件的使用，我们发现我们仅仅是在执行操作之前设置了一句 PageHelper.startPage(1, 5); 并没有做其他操作，也就是没有改变任何其他的业务代码。这就是它的优点，那么我再来看下他的实现原理

3.2 实现原理剖析

在PageHelper中，肯定有提供Interceptor的实现类，通过源码我们可以发现是PageHelper，而且我们也可以看到在该方法头部添加的注解，声明了该拦截器拦截的是Executor的query方法

```

import ...  

/**  

 * Mybatis - 通用分页拦截器  

 *  

 * @author liuzh/abel533/isean533  

 * @version 3.3.0  

 * 项目地址 : http://git.oschina.net/free/Mybatis_PageHelper  

*/  

@Intercepts(@Signature(type = Executor.class, method = "query", args = {MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class}))  

public class PageHelper implements Interceptor {  

    //sql工具类  

    private SqlUtil sqlUtil;  

    //属性参数信息  

    private Properties properties;  

    //配置对象方式  

    private SqlUtilConfig sqlUtilConfig;  

    //自动获取dialect,如果没有setProperties或setSqlUtilConfig,也可以正常进行  

    private boolean autoDialect = true;  

    //运行时自动获取dialect  

    private boolean autoRuntimeDialect;  

    //多数据源时,获取jdbcUrl后是否关闭数据源  

    private boolean closeConn = true;  

    //缓存  

    private Map<String, SqlUtil> urlSqlUtilMap = new ConcurrentHashMap<>();  

    private ReentrantLock lock = new ReentrantLock();  

    ...  

    /**  

     * 获得任意查询方法的count总数  

     *  

     * @param proxy  

     * @param method  

     * @param args  

     * @return
    
```

然后当我们要执行查询操作的时候，我们知道 Executor.query() 方法的执行本质上是执行 Executor 的代理对象的方法。先来看下Plugin中的invoke方法

```

    /**
     * 代理对象方法被调用时执行的代码
     * @param proxy
     * @param method
     * @param args
     * @return
    
```

```

    * @throws Throwable
    */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        // 获取当前方法所在类或接口中，可被当前Interceptor拦截的方法
        Set<Method> methods = signatureMap.get(method.getDeclaringClass());
        if (methods != null && methods.contains(method)) {
            // 当前调用的方法需要被拦截 执行拦截操作
            return interceptor.intercept(new Invocation(target, method, args));
        }
        // 不需要拦截 则调用 目标对象中的方法
        return method.invoke(target, args);
    } catch (Exception e) {
        throw ExceptionUtil.unwrapThrowable(e);
    }
}

```

interceptor.intercept(new Invocation(target, method, args));方法的执行会进入到 PageHelper 的 intercept 方法中

```

/**
 * Mybatis拦截器方法
 *
 * @param invocation 拦截器入参
 * @return 返回执行结果
 * @throws Throwable 抛出异常
 */
public Object intercept(Invocation invocation) throws Throwable {
    if (autoRuntimeDialect) {
        SqlUtil sqlutil = getSqlutil(invocation);
        return sqlutil.processPage(invocation);
    } else {
        if (autoDialect) {
            initSqlUtil(invocation);
        }
        return sqlutil.processPage(invocation);
    }
}

```

intercept方法

```

/**
 * Mybatis拦截器方法
 *
 * @param invocation 拦截器入参
 * @return 返回执行结果
 * @throws Throwable 抛出异常
 */
public Object intercept(Invocation invocation) throws Throwable {
    if (autoRuntimeDialect) { // 多数据源
        SqlUtil sqlutil = getSqlutil(invocation);
        return sqlutil.processPage(invocation);
    } else { // 单数据源
        if (autoDialect) {

```

```
        initSqlUtil(invocation);
    }
    return sqlUtil.processPage(invocation);
}
}
```

在interceptor方法中首先会获取一个SqlUtils对象

SqlUtil：数据库类型专用sql工具类，一个数据库url对应一个SqlUtil实例，SqlUtil内有一个Parser对象，如果是mysql，它是MysqlParser，如果是oracle，它是OracleParser，这个Parser对象是SqlUtil不同实例的主要存在价值。执行count查询、设置Parser对象、执行分页查询、保存Page分页对象等功能，均由SqlUtil来完成。

initSqlUtil 方法

```
/**
 * 初始化sqlUtil
 *
 * @param invocation
 */
public synchronized void initSqlUtil(Invocation invocation) {
    if (this.sqlUtil == null) {
        this.sqlUtil = getSqlUtil(invocation);
        if (!autoRuntimeDialect) {
            properties = null;
            sqlUtilConfig = null;
        }
        autoDialect = false;
    }
}
```

getSqlUtil方法

```
/**
 * 根据datasource创建对应的sqlUtil
 *
 * @param invocation
 */
public SqlUtil getSqlUtil(Invocation invocation) {
    MappedStatement ms = (MappedStatement) invocation.getArgs()[0];
    //改为对dataSource做缓存
    DataSource dataSource =
        ms.getConfiguration().getEnvironment().getDataSource();
    String url = getUrl(dataSource);
    if (urlSqlUtilMap.containsKey(url)) {
        return urlSqlUtilMap.get(url);
    }
    try {
        lock.lock();
        if (urlSqlUtilMap.containsKey(url)) {
            return urlSqlUtilMap.get(url);
        }
        if (StringUtil.isEmpty(url)) {
            throw new RuntimeException("无法自动获取jdbcUrl，请在分页插件中配置
dialect参数!");
        }
        String dialect = Dialect.fromJdbcUrl(url);
```

```
        if (dialect == null) {
            throw new RuntimeException("无法自动获取数据库类型，请通过dialect参数指定!");
        }
        // 创建SqlUtil
        SqlUtil sqlutil = new SqlUtil(dialect);
        if (this.properties != null) {
            sqlutil.setProperties(properties);
        } else if (this.sqlUtilConfig != null) {
            sqlutil.setSqlUtilConfig(this.sqlUtilConfig);
        }
        urlSqlUtilMap.put(url, sqlutil);
        return sqlutil;
    } finally {
        lock.unlock();
    }
}
```

查看SqlUtil的构造方法

```
/**
 * 构造方法
 *
 * @param strDialect
 */
public SqlUtil(String strDialect) {
    if (strDialect == null || "".equals(strDialect)) {
        throw new IllegalArgumentException("Mybatis分页插件无法获取dialect参数!");
    }
    Exception exception = null;
    try {
        Dialect dialect = Dialect.of(strDialect);
        // 根据方言创建对应的解析器
        parser = AbstractParser.newParser(dialect);
    } catch (Exception e) {
        exception = e;
        // 异常的时候尝试反射，允许自己写实现类传递进来
        try {
            Class<?> parserClass = Class.forName(strDialect);
            if (Parser.class.isAssignableFrom(parserClass)) {
                parser = (Parser) parserClass.newInstance();
            }
        } catch (ClassNotFoundException ex) {
            exception = ex;
        } catch (InstantiationException ex) {
            exception = ex;
        } catch (IllegalAccessException ex) {
            exception = ex;
        }
    }
    if (parser == null) {
        throw new RuntimeException(exception);
    }
}
```

创建的解析器

```
public static Parser newParser(Dialect dialect) {  
    Parser parser = null;  
    switch (dialect) {  
        case mysql:  
        case mariadb:  
        case sqlite:  
            parser = new MysqlParser();  
            break;  
        case oracle:  
            parser = new OracleParser();  
            break;  
        case hsqldb:  
            parser = new HsqldbParser();  
            break;  
        case sqlserver:  
            parser = new SqlServerParser();  
            break;  
        case sqlserver2012:  
            parser = new SqlServer2012Dialect();  
            break;  
        case db2:  
            parser = new Db2Parser();  
            break;  
        case postgresql:  
            parser = new PostgreSQLParser();  
            break;  
        case informix:  
            parser = new InformixParser();  
            break;  
        case h2:  
            parser = new H2Parser();  
            break;  
        default:  
            throw new RuntimeException("分页插件" + dialect + "方言错误!");  
    }  
    return parser;  
}
```

我们可以看到不同的数据库方言，创建了对应的解析器。

然后我们再回到前面的interceptor方法中继续，查看sqlUtil.processPage(invocation);方法

```
private Object _processPage(Invocation invocation) throws Throwable {  
    final Object[] args = invocation.getArgs();  
    Page page = null;  
    //支持方法参数时，会先尝试获取Page  
    if (supportMethodsArguments) {  
        page = getPage(args); // 有通过ThreadLocal来获取分页数据信息  
    }  
    //分页信息  
    RowBounds rowBounds = (RowBounds) args[2];  
    //支持方法参数时，如果page == null就说明没有分页条件，不需要分页查询  
    if ((supportMethodsArguments && page == null)  
        //当不支持分页参数时，判断LocalPage和RowBounds判断是否需要分页
```

```

    || (!supportMethodsArguments && sqlUtil.getLocalPage() == null)
&& rowBounds == RowBounds.DEFAULT)) {
    return invocation.proceed();
} else {
    //不支持分页参数时, page==null, 这里需要获取
    if (!supportMethodsArguments && page == null) {
        page = getPage(args);
    }
    return doProcessPage(invocation, page, args);
}
}

```

doProcessPage进入该方法

```

private Page doProcessPage(Invocation invocation, Page page, Object[] args)
throws Throwable {
    //保存RowBounds状态
    RowBounds rowBounds = (RowBounds) args[2];
    //获取原始的ms
    MappedStatement ms = (MappedStatement) args[0];
    //判断并处理为PageSqlSource
    if (!isPageSqlSource(ms)) {
        processMappedStatement(ms);
    }
    //设置当前的parser, 后面每次使用前都会set, ThreadLocal的值不会产生不良影响
    ((PageSqlSource)ms.getSqlSource()).setParser(parser);
    try {
        //忽略RowBounds-否则会进行Mybatis自带的内存分页
        args[2] = RowBounds.DEFAULT;
        //如果只进行排序 或 pageSizeZero的判断
        if (isQueryOnly(page)) {
            return doQueryOnly(page, invocation);
        }

        //简单的通过total的值来判断是否进行count查询
        if (page.isCount()) {
            page.setCountSignal(Boolean.TRUE);
            //替换MS
            args[0] = msCountMap.get(ms.getId());
            //查询总数
            Object result = invocation.proceed();
            //还原ms
            args[0] = ms;
            //设置总数
            page.setTotal((Integer)((List)result).get(0));
            if (page.getTotal() == 0) {
                return page;
            }
        } else {
            page.setTotal(-1);
        }
        //pageSize>0的时候执行分页查询, pageSize<=0的时候不执行相当于可能只返回了一个
        count
        if (page.getPageSize() > 0 &&
            ((rowBounds == RowBounds.DEFAULT && page.getPageNum() > 0)
            || rowBounds != RowBounds.DEFAULT)) {
            //将参数中的Mappedstatement替换为新的qs
        }
    }
}

```

```

        page.setCountSignal(null);
        BoundSql boundSql = ms.getBoundSql(args[1]);
        // 在 invocation中绑定 分页的数据
        args[1] = parser setPageParameter(ms, args[1], boundSql, page);
        page.setCountSignal(Boolean.FALSE);
        //执行分页查询
        Object result = invocation.proceed();
        //得到处理结果
        page.addAll((List) result);
    }
} finally {
    ((PageSqlSource)ms.getSqlSource()).removeParser();
}

//返回结果
return page;
}

```

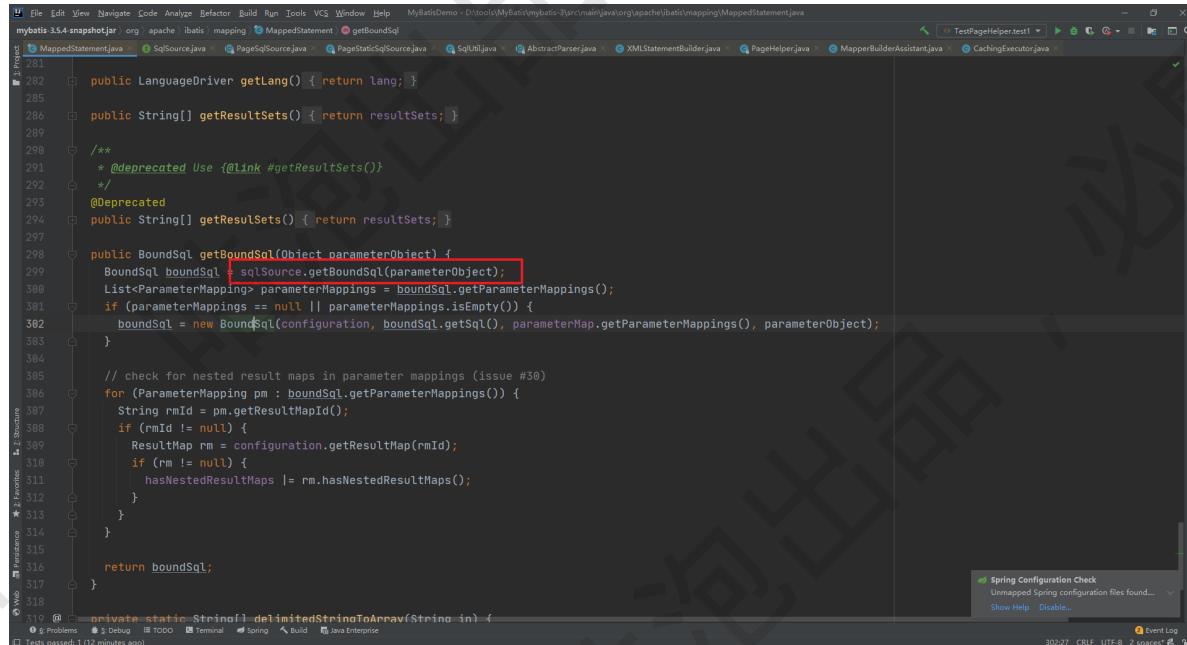
invocation.proceed();方法会进入 CachingExecutor中的query方法。

```

@Override
public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {
    // 获取SQL
    BoundSql boundSql = ms.getBoundSql(parameterObject);
    // 创建CacheKey: 什么样的SQL是同一条SQL? >>
    CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql);
    return query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
}

```

ms.getBoundSql方法中会完成分页SQL的绑定



```

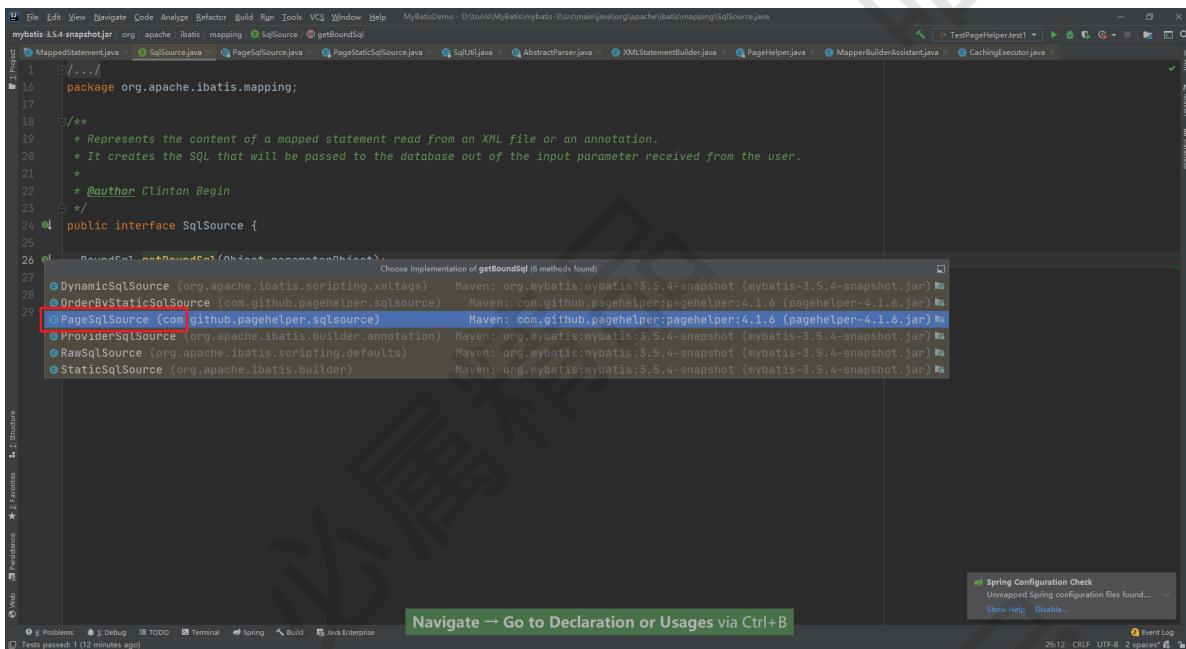
    public BoundSql getBoundSql(Object parameterObject) {
        BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
        List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
        if (parameterMappings == null || parameterMappings.isEmpty()) {
            boundSql = new BoundSql(configuration, boundSql.getSql(), parameterMap.getParameterMappings(), parameterObject);
        }

        // check for nested result maps in parameter mappings (issue #50)
        for (ParameterMapping pm : boundSql.getParameterMappings()) {
            String rmid = pm.getResultMapId();
            if (rmid != null) {
                ResultMap rm = configuration.getResultMap(rmid);
                if (rm != null) {
                    hasNestedResultMaps |= rm.hasNestedResultMaps();
                }
            }
        }

        return boundSql;
    }
}

```

进入 PageSqlSource 的getBoundSql方法中



```
    /**
     * 获取BoundsSql
     *
     * @param parameterObject
     * @return
     */
    @Override
    public BoundsSql getBoundSql(Object parameterObject) {
        Boolean count = getCount();
        if (count == null) {
            return getDefaultBoundSql(parameterObject);
        } else if (count) {
            return getCountBoundSql(parameterObject);
        } else {
            return getPageBoundSql(parameterObject);
        }
    }
```

然后进入getPageBoundSql获取分页的SQL语句，在这个方法中大家也可以发现查询总的记录数的SQL生成

```
    @Override
    protected BoundsSql getPageBoundSql(Object parameterObject) {
        String tempSql = sql;
        String orderBy = PageHelper.getOrderBy();
        if (orderBy != null) {
            tempSql = OrderByParser.convertToOrderBySql(sql, orderBy);
        }
        tempSql = localParser.get().getPageSql(tempSql);
        return new BoundsSql(configuration, tempSql,
                localParser.get().getPageParameterMapping(configuration,
                original.getBoundSql(parameterObject)), parameterObject);
    }
```

最终在这个方法中生成了对应数据库的分页语句

```

    ...
    return new BoundSql(configuration, tempSql, parameterMappings, parameterObject);
}

@Override
protected BoundSql getCountBoundSql(Object parameterObject) {
    return new BoundSql(configuration, localParser.get().getCountSql(sql), parameterMappings, parameterObject);
}

@Override
protected BoundSql getPageBoundSql(Object parameterObject) { parameterObject: size = 3
    String tempSql = sql; tempSql = "select * from t_user limit ?,?"
    String orderBy = PageHelper.getOrderby(); orderBy: null
    if (orderBy != null) {
        tempSql = OrderByParser.convertToOrderBySql(sql, orderBy); sql: "select * from t_user" orderBy: null
    }
    tempSql = localParser.get().getPageSql(tempSql);
    return new BoundSql(configuration, tempSql, localParser.get().getPageParameterMapping(configuration, originalSql));
}
}

```

4.应用场景分析

作用	描述	实现方式
水平分表	一张费用表按月度拆分为12张表。fee_202001-202012。当查询条件出现月度(tran_month)时，把select语句中的逻辑表名修改为对应的月份表。	对query update方法进行拦截在接口上添加注解，通过反射获取接口注解，根据注解上配置的参数进行分表，修改原SQL，例如id取模，按月分表
数据脱敏	手机号和身份证号在数据库完整存储。但是返回给用户，屏蔽手机号的中间四位。屏蔽身份证号中的出生日期。	query——对结果集脱敏
菜单权限控制	不同的用户登录，查询菜单权限表时获得不同的结果，在前端展示不同的菜单	对query方法进行拦截在方法上添加注解，根据权限配置，以及用户登录信息，在SQL上加上权限过滤条件
黑白名单	有些SQL语句在生产环境中是不允许执行的，比如like %%	对Executor的update和query方法进行拦截，将拦截的SQL语句和黑名单进行比较，控制SQL语句的执行
全局唯一ID	在高并发的环境下传统的生成ID的方式不太适用，这时我们就需要考虑其他方式了	创建插件拦截Executor的insert方法，通过UUID或者雪花算法来生成ID，并修改SQL中的插入信息

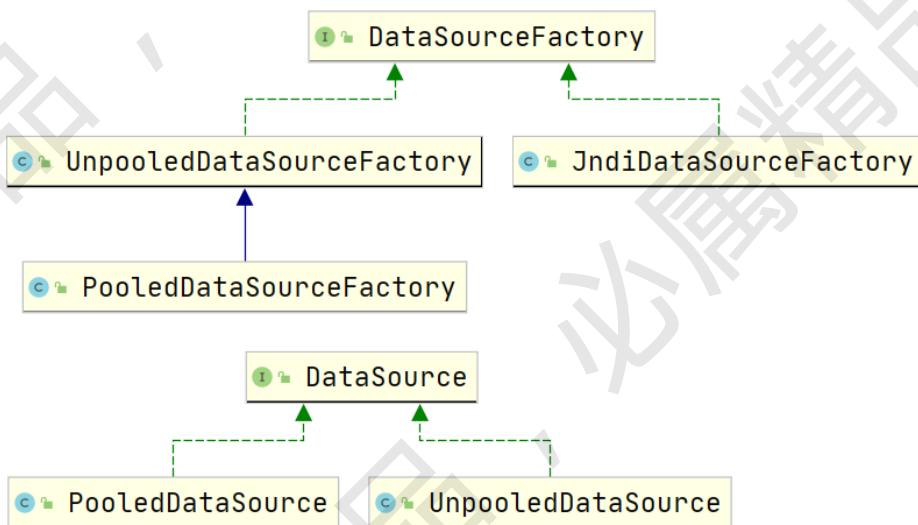
二、事务管理

在本节中我们来介绍下MyBatis中的事务管理，而介绍事务管理的话不可避免的要接触到DataSource的内容，所以我们会分别来介绍DataSource和Transaction两块内容。

1. DataSource

在数据持久层中，数据源是一个非常重要的组件，其性能直接关系到整个数据持久层的性能，在实际开发中我们常用的数据源有 Apache Common DBCP，C3P0，Druid 等，MyBatis不仅可以集成第三方数据源，还提供的有自己实现的数据源。

在MyBatis中提供了两个 javax.sql.DataSource 接口的实现，分别是 PooledDataSource 和 UnpooledDataSource .



1.1 DataSourceFactory

DataSourceFactory是用来创建DataSource对象的，接口中声明了两个方法，作用如下

```
public interface DataSourceFactory {
    // 设置 DataSource 的相关属性，一般紧跟在初始化完成之后
    void setProperties(Properties props);

    // 获取 DataSource 对象
    DataSource getDataSource();
}
```

DataSourceFactory接口的两个具体实现是 `UnpooledDataSourceFactory` 和 `PooledDataSourceFactory` 这两个工厂对象的作用通过名称我们也能发现是用来创建不带连接池的数据源对象和创建带连接池的数据源对象，先来看下 `UnpooledDataSourceFactory` 中的方法

```
/**
 * 完成对 UnpooledDataSource 的配置
 * @param properties 封装的有 DataSource 所需要的相关属性信息
```

```
/*
@Override
public void setProperties(Properties properties) {
    Properties driverProperties = new Properties();
    // 创建 DataSource 对应的 MetaObject 对象
    MetaObject metaDataSource = SystemMetaObject.forObject(dataSource);
    // 遍历 Properties 集合, 该集合中配置了数据源需要的信息
    for (Object key : properties.keySet()) {
        String propertyName = (String) key; // 获取属性名称
        if (propertyName.startsWith(DRIVER_PROPERTY_PREFIX)) {
            // 以 "driver." 开头的配置项是对 DataSource 的配置
            String value = properties.getProperty(propertyName);

            driverProperties.setProperty(propertyName.substring(DRIVER_PROPERTY_PREFIX_LENGTH), value);
        } else if (metaDataSource.hasSetter(propertyName)) {
            // 有该属性的 setter 方法
            String value = (String) properties.get(propertyName);
            Object convertedValue = convertValue(metaDataSource, propertyName, value);
            // 设置 DataSource 的相关属性值
            metaDataSource.setValue(propertyName, convertedValue);
        } else {
            throw new DataSourceException("Unknown DataSource property: " +
                propertyName);
        }
    }
    if (driverProperties.size() > 0) {
        // 设置 DataSource.driverProperties 的属性值
        metaDataSource.setValue("driverProperties", driverProperties);
    }
}
```

UnpooledDataSourceFactory的getDataSource方法实现比较简单，直接返回DataSource属性记录的UnpooledDataSource对象

1.2 UnpooledDataSource

UnpooledDataSource是DataSource接口的其中一个实现，但是UnpooledDataSource并没有提供数据库连接池的支持，我们来看下他的具体实现吧

声明的相关属性信息

```
private ClassLoader driverClassLoader; // 加载Driver的类加载器
private Properties driverProperties; // 数据库连接驱动的相关信息
// 缓存所有已注册的数据库连接驱动
private static Map<String, Driver> registeredDrivers = new ConcurrentHashMap<>();
{
    private String driver; // 驱动
    private String url; // 数据库 url
    private String username; // 账号
    private String password; // 密码

    private Boolean autoCommit; // 是否自动提交
    private Integer defaultTransactionIsolationLevel; // 事务隔离级别
    private Integer defaultNetworkTimeout;
}
```

然后在静态代码块中完成了 Driver 的复制

```
static {
    // 从 DriverManager 中获取 Drivers
    Enumeration<Driver> drivers = DriverManager.getDrivers();
    while (drivers.hasMoreElements()) {
        Driver driver = drivers.nextElement();
        // 将获取的 Driver 记录到 Map 集合中
        registeredDrivers.put(driver.getClass().getName(), driver);
    }
}
```

UnpooledDataSource 中获取 Connection 的方法最终都会调用 doGetConnection() 方法。

```
private Connection doGetConnection(Properties properties) throws SQLException
{
    // 初始化数据库驱动
    initializeDriver();
    // 创建真正的数据库连接
    Connection connection = DriverManager.getConnection(url, properties);
    // 配置 Connection 的自动提交和事务隔离级别
    configureConnection(connection);
    return connection;
}
```

1.3 PooledDataSource

有开发经验的小伙伴都知道，在操作数据库的时候数据库连接的创建过程是非常耗时的，数据库能够建立的连接数量也是非常有限的，所以数据库连接池的使用是非常重要的，使用数据库连接池会给我们带来很多好处，比如可以实现数据库连接的重用，提高响应速度，防止数据库连接过多造成数据库假死，避免数据库连接泄漏等等。

首先来看下声明的相关的属性

```
// 管理状态
private final PoolState state = new Poolstate(this);

// 记录UnpooledDataSource，用于生成真实的数据库连接对象
```

```

private final UnpooledDataSource datasource;

// OPTIONAL CONFIGURATION FIELDS
protected int poolMaximumActiveConnections = 10; // 最大活跃连接数
protected int poolMaximumIdleConnections = 5; // 最大空闲连接数
protected int poolMaximumCheckoutTime = 20000; // 最大checkout时间
protected int poolTimeToWait = 20000; // 无法获取连接的线程需要等待的时长
protected int poolMaximumLocalBadConnectionTolerance = 3; //
protected String poolPingQuery = "NO PING QUERY SET"; // 测试的SQL语句
protected boolean poolPingEnabled; // 是否允许发送测试SQL语句
// 当连接超过 poolPingConnectionsNotUsedFor毫秒未使用时，会发送一次测试SQL语句，检测连接是否正常
protected int poolPingConnectionsNotUsedFor;
// 根据数据库URL，用户名和密码生成的一个hash值。
private int expectedConnectionTypeCode;

```

然后重点来看下 `getConnection` 方法，该方法是用来给调用者提供 `Connection` 对象的。

```

@Override
public Connection getConnection() throws SQLException {
    return popConnection(dataSource.getUsername(),
    dataSource.getPassword()).getProxyConnection();
}

```

我们会发现其中调用了 `popConnection` 方法，在该方法中返回的是 `PooledConnection` 对象，而 `PooledConnection` 对象实现了 `InvocationHandler` 接口，所以会使用到Java的动态代理，其中相关的属性为

```

private static final String CLOSE = "close";
private static final Class<?>[] IFACES = new Class<?>[] { Connection.class };

private final int hashCode;
private final PooledDataSource dataSource;
// 真正的数据库连接
private final Connection realConnection;
// 数据库连接的代理对象
private final Connection proxyConnection;
private long checkoutTimestamp; // 从连接池中取出该连接的时间戳
private long createdTimestamp; // 该连接创建的时间戳
private long lastUsedTimestamp; // 最后一次被使用的时间戳
private int connectionTypeCode; // 又数据库URL、用户名和密码计算出来的hash值，可用于标识该连接所在的连接池
// 连接是否有效的标志
private boolean valid;

```

重点关注下 `invoke` 方法

```

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    String methodName = method.getName();
    if (CLOSE.equals(methodName)) {
        // 如果是 close 方法被执行则将连接放回连接池中，而不是真正的关闭数据库连接
        dataSource.pushConnection(this);
    }
}

```

```

        return null;
    }
    try {
        if (!Object.class.equals(method.getDeclaringClass())) {
            // issue #579 toString() should never fail
            // throw an SQLException instead of a Runtime
            // 通过上面的 valid 字段来检测 连接是否有效
            checkConnection();
        }
        // 调用真正数据库连接对象的对应方法
        return method.invoke(realConnection, args);
    } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
    }
}

```

还有就是前面提到的 PoolState 对象，它主要是用来管理 PooledConnection 对象状态的组件，通过两个 ArrayList 集合分别管理空闲状态的连接和活跃状态的连接，定义如下：

```

protected PooledDataSource dataSource;
// 空闲的连接
protected final List<PooledConnection> idleConnections = new ArrayList<>();
// 活跃的连接
protected final List<PooledConnection> activeConnections = new ArrayList<>();
protected long requestCount = 0; // 请求数据库连接的次数
protected long accumulatedRequestTime = 0; // 获取连接累计的时间
// CheckoutTime 表示应用从连接池中取出来，到归还连接的时长
// accumulatedCheckoutTime 记录了所有连接累计的CheckoutTime时长
protected long accumulatedCheckoutTime = 0;
// 当连接长时间没有归还连接时，会被认为该连接超时
// claimedOverdueConnectionCount 记录连接超时的个数
protected long claimedOverdueConnectionCount = 0;
// 累计超时时间
protected long accumulatedCheckoutTimeOfOverdueConnections = 0;
// 累计等待时间
protected long accumulatedWaitTime = 0;
// 等待次数
protected long hadToWaitCount = 0;
// 无效连接数
protected long badConnectionCount = 0;

```

再回到 popConnection 方法中来看

```

private PooledConnection popConnection(String username, String password)
throws SQLException {
    boolean countedWait = false;
    PooledConnection conn = null;
    long t = System.currentTimeMillis();
    int localBadConnectionCount = 0;

    while (conn == null) {
        synchronized (state) { // 同步
            if (!state.idleConnections.isEmpty()) { // 检测空闲连接
                // Pool has available connection 连接池中有空闲的连接
                conn = state.idleConnections.remove(0); // 获取连接
            }
        }
    }
}

```

```
        if (log.isDebugEnabled()) {
            log.debug("Checked out connection " + conn.getRealHashCode() + " from pool.");
        }
    } else { // 当前连接池 没有空闲连接
        // Pool does not have available connection
        if (state.activeConnections.size() < poolMaximumActiveConnections) {
            // 活跃数没有达到最大连接数 可以创建新的连接
            // Can create new connection 创建新的数据库连接
            conn = new PooledConnection(dataSource.getConnection(), this);
            if (log.isDebugEnabled()) {
                log.debug("Created connection " + conn.getRealHashCode() + ".");
            }
        } else { // 活跃数已经达到了最大数 不能创建新的连接
            // Cannot create new connection 获取最先创建的活跃连接
            PooledConnection oldestActiveConnection =
state.activeConnections.get(0);
            // 获取该连接的超时时间
            long longestCheckoutTime = oldestActiveConnection.getCheckoutTime();
            // 检查是否超时
            if (longestCheckoutTime > poolMaximumCheckoutTime) {
                // Can claim overdue connection 对超时连接的信息进行统计
                state.claimedOverdueConnectionCount++;
                state.accumulatedCheckoutTimeOfOverdueConnections +=
longestCheckoutTime;
                state.accumulatedCheckoutTime += longestCheckoutTime;
                // 将超时连接移除 activeConnections
                state.activeConnections.remove(oldestActiveConnection);
                if (!oldestActiveConnection.getRealConnection().getAutoCommit()) {
                    // 如果超时连接没有提交 则自动回滚
                    try {
                        oldestActiveConnection.getRealConnection().rollback();
                    } catch (SQLException e) {
                        /*
                         * Just log a message for debug and continue to execute the
                         * statement like nothing happened.
                         * Wrap the bad connection with a new PooledConnection, this
                         * to not interrupt current executing thread and give current
                         * chance to join the next competition for another valid/good
                         * connection. At the end of this loop, bad {@link @conn} will
                         * be set as null.
                         */
                        log.debug("Bad connection. Could not roll back");
                    }
                }
            }
            // 创建 PooledConnection, 但是数据库中的真正连接并没有创建
            conn = new
PooledConnection(oldestActiveConnection.getRealConnection(), this);

            conn.setCreatedTimestamp(oldestActiveConnection.getCreatedTimestamp());
            conn.setLastUsedTimestamp(oldestActiveConnection.getLastUsedTimestamp());
            // 将超时的 PooledConnection 设置为无效
            oldestActiveConnection.invalidate();
        }
    }
}
```

```
        if (log.isDebugEnabled()) {
            log.debug("Claimed overdue connection " + conn.getRealHashCode()
+ ".");
        }
    } else {
        // Must wait 无空闲连接，无法创建新连接和无超时连接 那就只能等待
        try {
            if (!countedWait) {
                state.hadTowaitCount++; // 统计等待次数
                countedWait = true;
            }
            if (log.isDebugEnabled()) {
                log.debug("Waiting as long as " + poolTimeTowait + "
milliseconds for connection.");
            }
            long wt = System.currentTimeMillis();
            state.wait(poolTimeTowait); // 阻塞等待
            // 统计累计的等待时间
            state.accumulatedWaitTime += System.currentTimeMillis() - wt;
        } catch (InterruptedException e) {
            break;
        }
    }
}

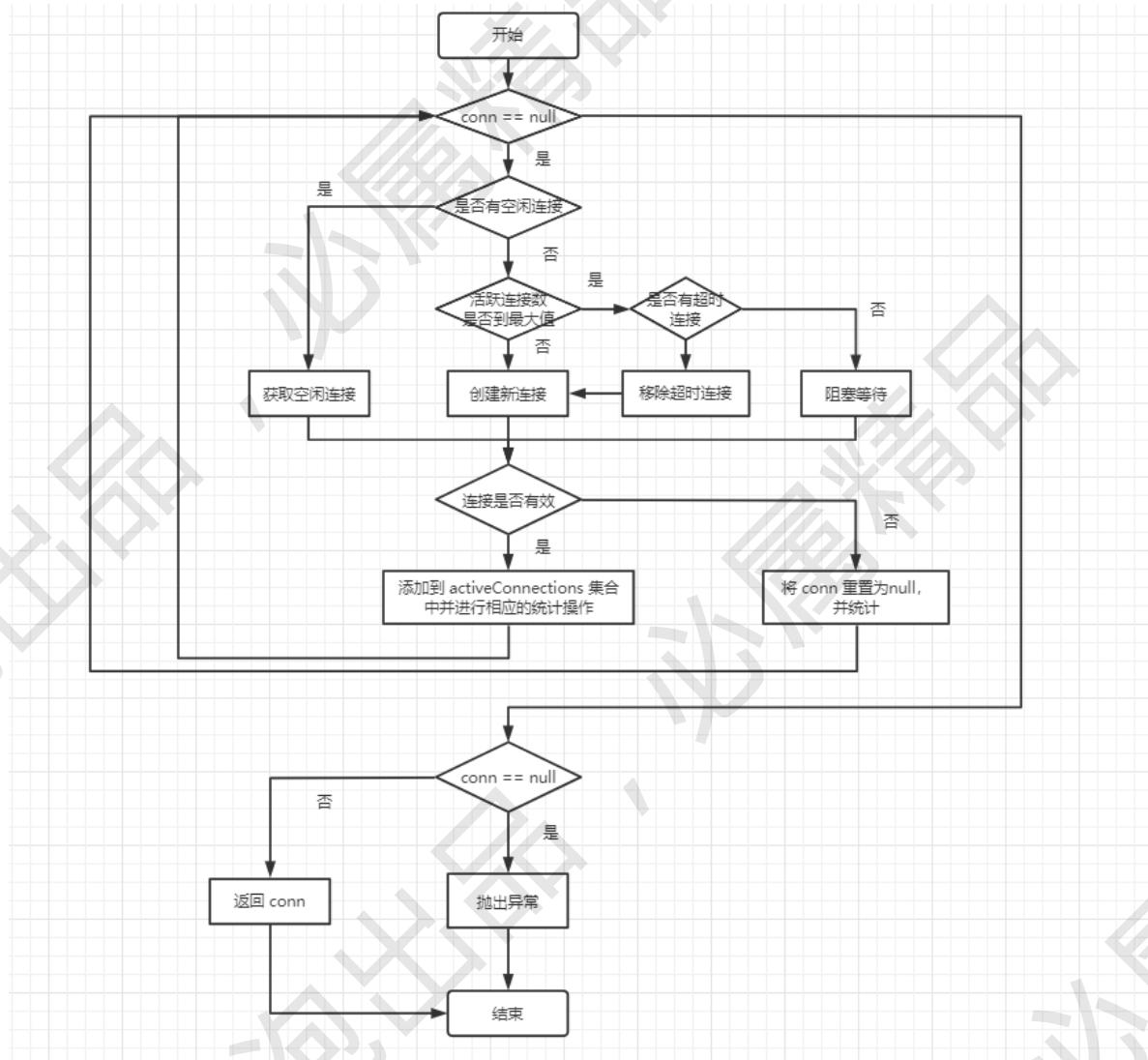
if (conn != null) {
    // ping to server and check the connection is valid or not
    // 检查 PooledConnection 是否有效
    if (conn.isValid()) {
        if (!conn.getRealConnection().getAutoCommit()) {
            conn.getRealConnection().rollback();
        }
        // 配置 PooledConnection 的相关属性

        conn.setConnectionTypeCode(assembleConnectionTypeCode(dataSource.getUrl(),
username, password));
        conn.setCheckoutTimestamp(System.currentTimeMillis());
        conn.setLastUsedTimestamp(System.currentTimeMillis());
        state.activeConnections.add(conn);
        state.requestCount++; // 进行相关的统计
        state.accumulatedRequestTime += System.currentTimeMillis() - t;
    } else {
        if (log.isDebugEnabled()) {
            log.debug("A bad connection (" + conn.getRealHashCode() + ") was
returned from the pool, getting another connection.");
        }
        state.badConnectionCount++;
        localBadConnectionCount++;
        conn = null;
        if (localBadConnectionCount > (poolMaximumIdleConnections +
poolMaximumLocalBadConnectionTolerance)) {
            if (log.isDebugEnabled()) {
                log.debug("PooledDataSource: Could not get a good connection to
the database.");
            }
            throw new SQLException("PooledDataSource: Could not get a good
connection to the database.");
        }
    }
}
```

```
    }  
}  
}
```

```
}
```

为了更好的理解代码的含义，我们绘制了对应的流程图



然后我们来看下当我们从连接池中使用完成了数据库的相关操作后，是如何来关闭连接的呢？通过前面的 invoke 方法的介绍其实我们能够发现，当我们执行代理对象的 close 方法的时候其实是执行的 pushConnection 方法。

```

  * @param method - the method to be executed
  * @param args - the parameters to be passed to the method
  * @see java.lang.reflect.InvocationHandler#invoke(Object, java.lang.reflect.Method, Object[])
  */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String methodName = method.getName();
    if (!CLOSE.equals(methodName)) {
        // 如果是 close 方法被执行则将连接放回连接池中, 而不是真正的关闭数据库连接
        dataSource.pushConnection(this);
        return null;
    }
    try {
        if (!Object.class.equals(method.getDeclaringClass())) {
            // issue #579 toString() should never fail
            // throw an SQLException instead of a Runtime
            // 通过上面的 valid 字段来检测连接是否有效
            checkConnection();
        }
        // 调用真正数据库连接对象的对应方法
        return method.invoke(realConnection, args);
    } catch (Throwable t) {
        throw ExceptionUtil.unwrapThrowable(t);
    }
}

private void checkConnection() throws SQLException {
    if (!valid) {
        throw new SQLException("Error accessing PooledConnection. Connection is invalid.");
    }
}

```

具体的实现代码为

```

protected void pushConnection(PooledConnection conn) throws SQLException {

    synchronized (state) {
        // 从 activeConnections 中移除 PooledConnection 对象
        state.activeConnections.remove(conn);
        if (conn.isValid()) { // 检测 连接是否有效
            if (state.idleConnections.size() < poolMaximumIdleConnections // 是否达到
上限
                && conn.getConnectionTypeCode() == expectedConnectionTypeCode // 该
PooledConnection 是否为该连接池的连接
            ) {
                state.accumulatedCheckoutTime += conn.getCheckoutTime(); // 累计
checkout 时长
                if (!conn.getRealConnection().getAutoCommit()) { // 回滚未提交的事务
                    conn.getRealConnection().rollback();
                }
                // 为返还连接创建新的 PooledConnection 对象
                PooledConnection newConn = new
                PooledConnection(conn.getRealConnection(), this);
                // 添加到 空闲连接集合中
                state.idleConnections.add(newConn);
                newConn.setCreatedTimestamp(conn.getCreatedTimestamp());
                newConn.setLastUsedTimestamp(conn.getLastUsedTimestamp());
                conn.invalidate(); // 将原来的 PooledConnection 连接设置为无效
                if (log.isDebugEnabled()) {
                    log.debug("Returned connection " + newConn.getRealHashCode() + " to
pool.");
                }
                // 唤醒阻塞等待的线程
                state.notifyAll();
            } else { // 空闲连接达到上限或者 PooledConnection 不属于当前的连接池
                state.accumulatedCheckoutTime += conn.getCheckoutTime(); // 累计
checkout 时长
                if (!conn.getRealConnection().getAutoCommit()) {
                    conn.getRealConnection().rollback();
                }
                conn.getRealConnection().close(); // 关闭真正的数据库连接
            }
        }
    }
}

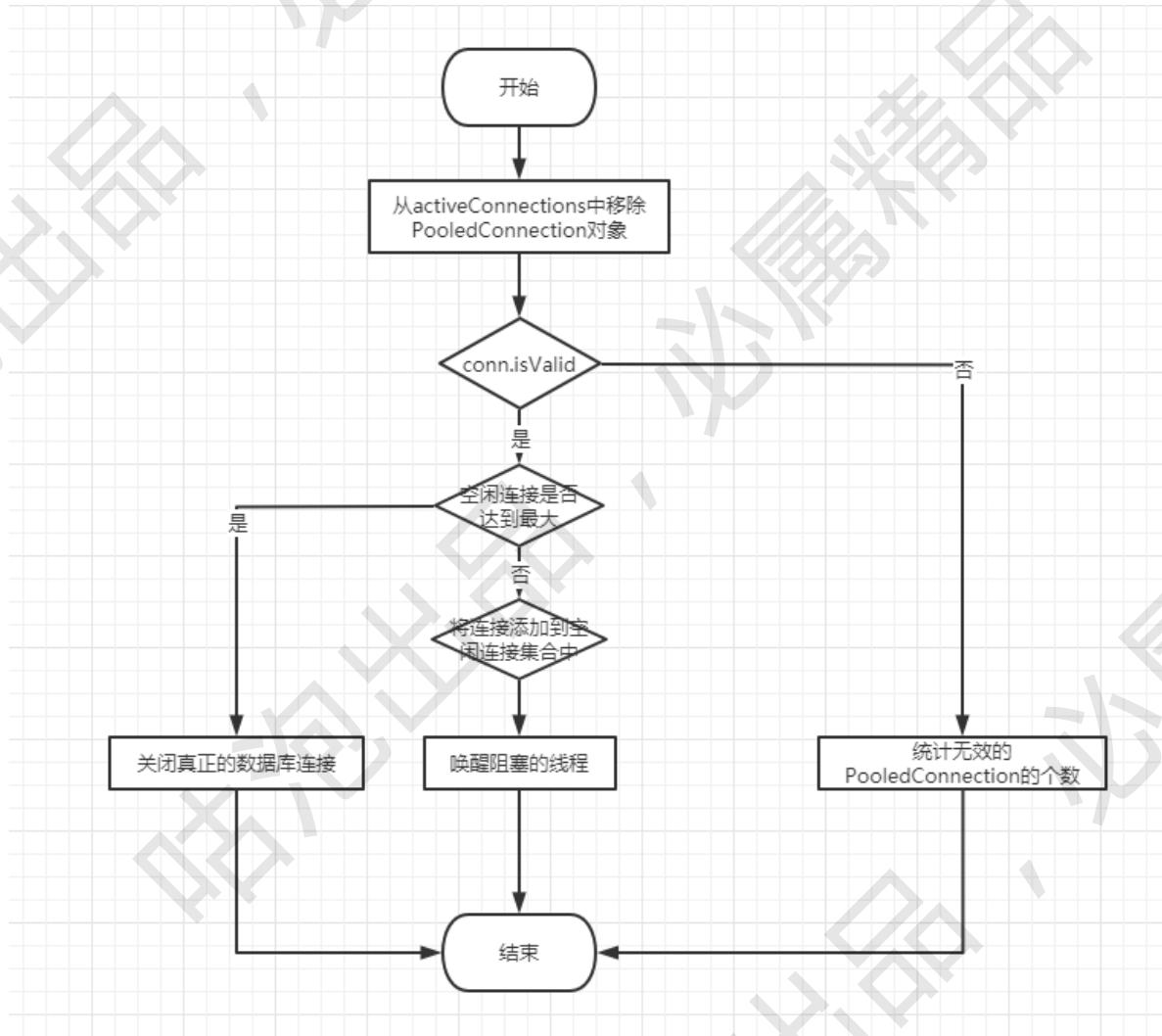
```

```

        if (log.isDebugEnabled()) {
            log.debug("Closed connection " + conn.getRealHashCode() + ".");
        }
        conn.invalidate(); // 设置 PooledConnection 无线
    }
} else {
    if (log.isDebugEnabled()) {
        log.debug("A bad connection (" + conn.getRealHashCode() + ") attempted
to return to the pool, discarding connection.");
    }
    state.badConnectionCount++; // 统计无效的 PooledConnection 对象个数
}
}
}
}

```

为了便于理解，我们同样的来绘制对应的流程图：



还有就是在源码中多处有看到 conn.isValid方法来检测连接是否有效

```

public boolean isValid() {
    return valid && realConnection != null && dataSource.pingConnection(this);
}

```

dataSource.pingConnection(this)中会真正的实现数据库的SQL执行操作

```

    if (result) {
        if (poolingEnabled) {
            if (poolingConnectionsNotUsedFor >= 0 && conn.getTimeElapsedSinceLastUse() > poolingConnectionsNotUsedFor) {
                try {
                    if (log.isDebugEnabled()) {
                        log.debug("Testing connection " + conn.getRealHashCode() + "...");
                    }
                    Connection realConn = conn.getRealConnection();
                    try (Statement statement = realConn.createStatement()) {
                        statement.executeQuery(poolPingQuery).close();
                    }
                    if (!realConn.getAutoCommit()) {
                        realConn.rollback();
                    }
                } catch (Exception e) {
                    log.warn("Execution of ping query " + poolPingQuery + " failed: " + e.getMessage());
                }
            }
            result = true;
        }
        if (log.isDebugEnabled()) {
            log.debug("Connection " + conn.getRealHashCode() + " is GOOD!");
        }
    } catch (Exception e) {
        log.warn("Execution of ping query " + poolPingQuery + " failed: " + e.getMessage());
    }
}

```

最后一点要注意的是在我们修改了任意的PooledDataSource中的属性的时候都会执行forceCloseAll来强制关闭所有的连接。

```

    this.poolMaximumLocalBadConnectionTolerance = poolMaximumLocalBadConnectionTolerance;

    /**
     * The maximum time a connection can be used before it *may* be given away again.
     *
     * @param poolMaximumCheckoutTime The maximum time
     */
    public void setPoolMaximumCheckoutTime(int poolMaximumCheckoutTime) {
        this.poolMaximumCheckoutTime = poolMaximumCheckoutTime;
        forceCloseAll();
    }

    /**
     * The time to wait before retrying to get a connection.
     *
     * @param poolTimeToWait The time to wait
     */
    public void setPoolTimeToWait(int poolTimeToWait) {
        this.poolTimeToWait = poolTimeToWait;
        forceCloseAll();
    }

    /**
     * The query to be used to check a connection.
     *
     * @param poolPingQuery The query
     */
    public void setPoolPingQuery(String poolPingQuery) {
        this.poolPingQuery = poolPingQuery;
        forceCloseAll();
    }
}

```

```

    /**
     * Closes all active and idle connections in the pool.
     */
    public void forceCloseAll() {
        synchronized (state) {
            // 更新当前的连接池标识
            expectedConnectionTypeCode =
            assembleConnectionTypeCode(dataSource.getUrl(), dataSource.getUsername(),
            dataSource.getPassword());
            for (int i = state.activeConnections.size(); i > 0; i--) { // 处理全部的活跃连接
                try {
                    // 获取获取的连接
                    PooledConnection conn = state.activeConnections.remove(i - 1);
                    conn.invalidate(); // 标识为无效连接
                    // 获取真实的数据库连接
                }
            }
        }
    }
}

```

```

        Connection realConn = conn.getRealConnection();
        if (!realConn.getAutoCommit()) {
            realConn.rollback(); // 回滚未处理的事务
        }
        realConn.close(); // 关闭真正的数据库连接
    } catch (Exception e) {
        // ignore
    }
}

// 同样的逻辑处理空闲的连接
for (int i = state.idleConnections.size(); i > 0; i--) {
    try {
        PooledConnection conn = state.idleConnections.remove(i - 1);
        conn.invalidate();

        Connection realConn = conn.getRealConnection();
        if (!realConn.getAutoCommit()) {
            realConn.rollback();
        }
        realConn.close();
    } catch (Exception e) {
        // ignore
    }
}
if (log.isDebugEnabled()) {
    log.debug("PooledDataSource forcefully closed/removed all connections.");
}
}

```

2.Transaction

在实际开发中，控制数据库事务是一件非常重要的工作，MyBatis使用Transaction接口对事务进行了抽象，定义的接口为

```

public interface Transaction {

    /**
     * Retrieve inner database connection. 获取对应的数据库连接
     * @return DataBase connection
     * @throws SQLException
     */
    Connection getConnection() throws SQLException;

    /**
     * Commit inner database connection. 提交事务
     * @throws SQLException
     */
    void commit() throws SQLException;

    /**
     * Rollback inner database connection. 回滚事务
     * @throws SQLException
     */
}

```

```
/*
void rollback() throws SQLException;

/**
 * Close inner database connection. 关闭连接
 * @throws SQLException
 */
void close() throws SQLException;

/**
 * Get transaction timeout if set. 获取事务超时时间
 * @throws SQLException
 */
Integer getTimeout() throws SQLException;

}
```

Transaction接口的实现有两个分别是 JdbcTransaction 和 ManagedTransaction两个

2.1 JdbcTransaction

JdbcTransaction 依赖于JDBC Connection来控制事务的提交和回滚，声明的相关的属性为

```
protected Connection connection; // 事务对应的数据库连接
protected DataSource dataSource; // 数据库连接所属的 数据源
protected TransactionIsolationLevel level; // 事务的隔离级别
protected boolean autoCommit; // 是否自动提交
```

在构造方法中会完成除了 Connection 属性外的另外三个属性的初始化，而Connection会延迟初始化，在我们执行getConnection方法的时候才会执行相关操作。源码比较简单，请自行查阅

2.2 ManagedTransaction

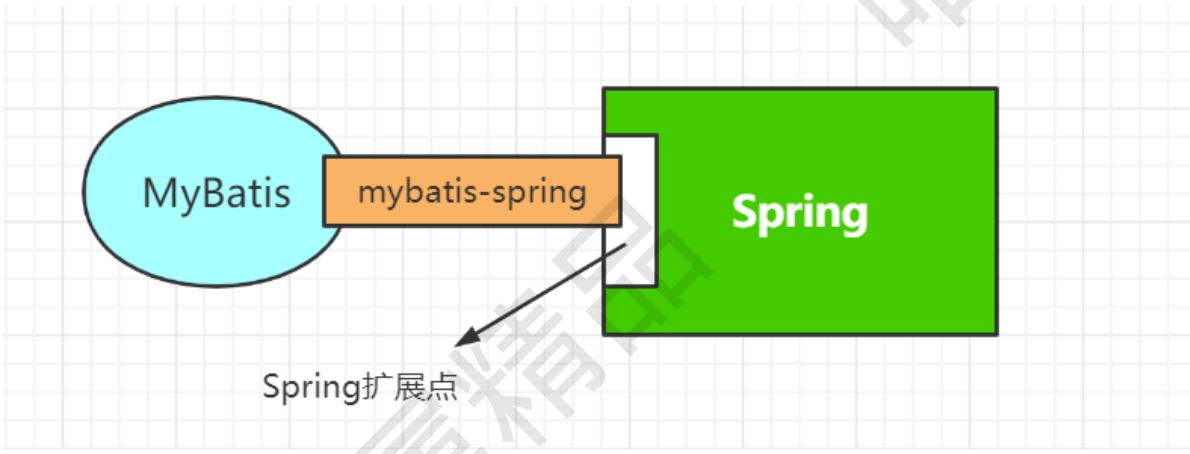
ManagedTransaction的实现更加的简单，它同样依赖 DataSource 字段来获取 Connection 对象，但是 commit方法和rollback方法都是空的，事务的提交和回滚都是依赖容器管理的。在实际开发中 MyBatis通常会和Spring集成，数据库的事务是交给Spring进行管理的，这个我们会在MyBatis整合 Spring中给大家介绍 SpringManagedTransaction。

三、MyBatis整合Spring

<http://mybatis.org/spring/zh/index.html>

1. MyBatis整合Spring实现

我们先来实现MyBatis和Spring的整合操作。



1.1 添加相关的依赖

```

<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.4</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.1.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.14</version>
</dependency>

```

1.2 配置文件

我们将MyBatis整合到Spring中，那么原来在MyBatis的很多配置我们都可以在Spring的配置文件中设置，我们可以给MyBatis的配置文件设置为空

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

</configuration>

```

添加Spring的配置文件，并在该文件中实现和Spring的整合操作

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <!-- 关联数据属性文件 -->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 开启扫描 -->
    <context:component-scan base-package="com.gupaoedu"/>

    <!-- 配置数据源 -->
    <bean class="com.alibaba.druid.pool.DruidDataSource"
        id="dataSource" >
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>
    <!-- 整合mybatis -->
    <bean class="org.mybatis.spring.SqlSessionFactoryBean"
        id="sqlSessionFactory" >
        <!-- 关联数据源 -->
        <property name="dataSource" ref="dataSource"/>
        <!-- 关联mybatis的配置文件 -->
        <property name="configLocation" value="classpath:mybatis-config-
spring.xml"/>
        <!-- 指定映射文件的位置 -->
        <property name="mapperLocations" value="classpath:mapper/*.xml" />
        <!-- 添加别名 -->
        <property name="typeAliasesPackage" value="com.gupaoedu.domain" />
    </bean>
    <!-- 配置扫描的路径 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer" >
        <property name="basePackage" value="com.gupaoedu.mapper"/>
    </bean>

```

```
</beans>
```

1.3 单元测试

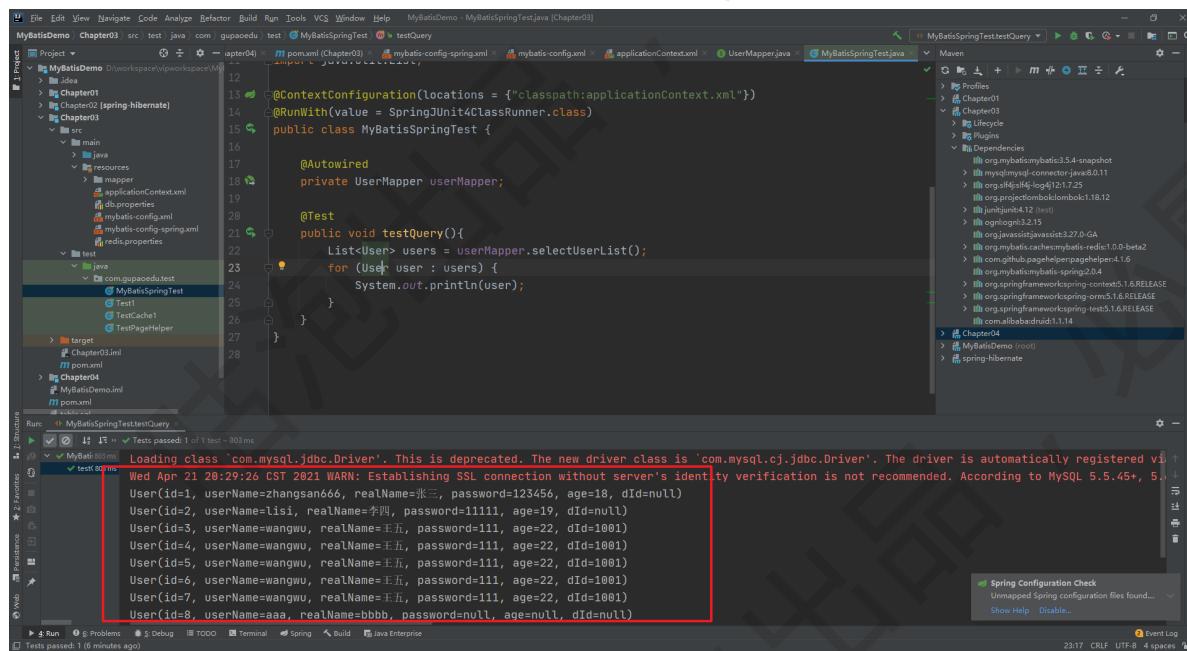
然后我们就可以通过测试来操作。如下：

```
@ContextConfiguration(locations = {"classpath:applicationContext.xml"})
@RunWith(value = SpringJUnit4ClassRunner.class)
public class MyBatisSpringTest {

    @Autowired
    private UserMapper userMapper;

    @Test
    public void testQuery(){
        List<User> users = userMapper.selectUserList();
        for (User user : users) {
            System.out.println(user);
        }
    }
}
```

查询到的结果



通过单元测试的代码我们可以发现，将MyBatis整合到Spring中后，原来操作的核心对象(SqlSessionFactory,SqlSession,getMapper)都不见了，使我们的开发更加的简洁。

2.MyBatis整合Spring的原理

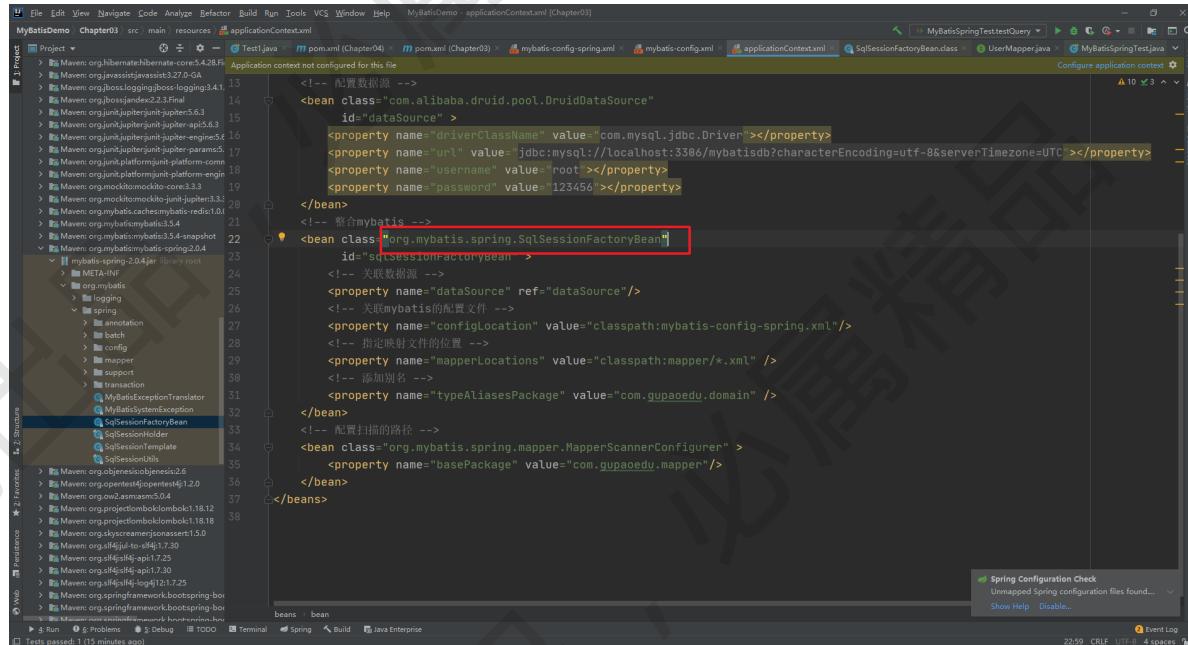
把MyBatis集成到Spring里面，是为了进一步简化MyBatis的使用，所以只是对MyBatis做了一些封装，并没有替换MyBatis的核心对象。也就是说：MyBatis jar包中的SqlSessionFactory、SqlSession、MapperProxy这些类都会用到。mybatis-spring.jar里面的类只是做了一些包装或者桥梁的工作。

只要我们弄明白了这三个对象是怎么创建的，也就理解了Spring继承MyBatis的原理。我们把它分成三步：

1. SqlSessionFactory在哪创建的。
2. SqlSession在哪创建的。
3. 代理类在哪创建的。

2.1 SqlSessionFactory

首先我们来看下在MyBatis整合Spring中SqlSessionFactory的创建过程，查看这步的入口在Spring的配置文件中配置整合的标签中



The screenshot shows the IntelliJ IDEA interface with the file `applicationContext.xml` open. The configuration for `SqlSessionFactoryBean` is highlighted with a red box. The code snippet is as follows:

```
<!-- 配置数据源 -->
<bean class="com.alibaba.druid.pool.DruidDataSource"
      id="dataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8&serverTimezone=UTC"/>
    <property name="username" value="root"/>
    <property name="password" value="123456"/>

```

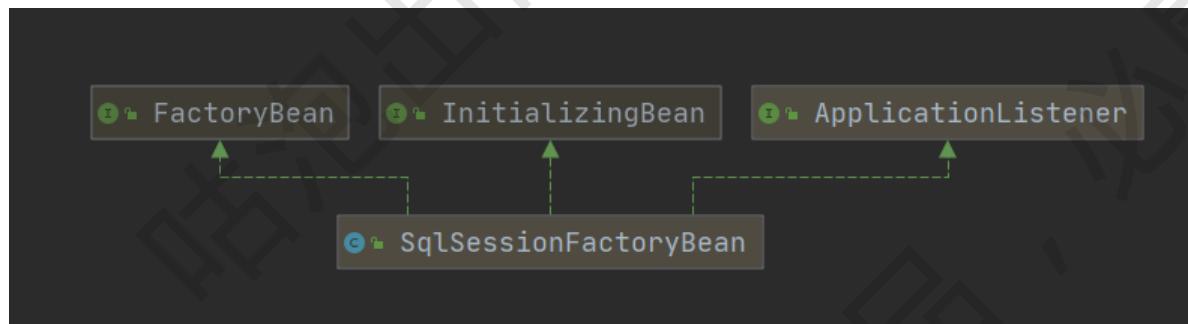
```
<!-- 整合mybatis -->
<bean class="org.mybatis.spring.SqlSessionFactoryBean"
      id="sqlSessionFactoryBean">
    <!-- 关联数据源 -->
    <property name="dataSource" ref="dataSource"/>
    <!-- 关联mybatis的配置文件 -->
    <property name="configLocation" value="classpath:mybatis-config-spring.xml"/>
    <!-- 指定映射文件的位置 -->
    <property name="mapperLocations" value="classpath:mapper/*.xml" />
    <!-- 添加别名 -->
    <property name="typeAliasesPackage" value="com.gupaoedu.domain" />

```

```
<!-- 配置扫描的路径 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.gupaoedu.mapper"/>
</bean>

```

我们进入`SqlSessionFactoryBean`中查看源码发现，其实现了`InitializingBean`、`FactoryBean`、`ApplicationListener`三个接口



对于这三个接口，学过Spring生命周期的小伙伴应该清楚他们各自的作用

接口	方法	作用
FactoryBean	<code>getObject()</code>	返回由FactoryBean创建的Bean实例
InitializingBean	<code>afterPropertiesSet()</code>	bean属性初始化完成后添加操作
ApplicationListener	<code>onApplicationEvent()</code>	对应用的时间进行监听

2.1.1 afterPropertiesSet

我们首先来看下 afterPropertiesSet 方法中的逻辑

```
public void afterPropertiesSet() throws Exception {  
    Assert.notNull(this.dataSource, "Property 'dataSource' is required");  
    Assert.notNull(this.sqlSessionFactoryBuilder, "Property  
'sqlSessionFactoryBuilder' is required");  
    Assert.state(this.configuration == null && this.configLocation == null ||  
this.configuration == null || this.configLocation == null, "Property  
'configuration' and 'configLocation' can not specified with together");  
    this.sqlSessionFactory = this.buildSqlSessionFactory();  
}
```

可以发现在afterPropertiesSet中直接调用了buildSqlSessionFactory方法来实现 sqlSessionFactory 对象的创建

```
protected SqlSessionFactory buildSqlSessionFactory() throws Exception {  
    // 解析全局配置文件的 XMLConfigBuilder 对象  
    XMLConfigBuilder xmlConfigBuilder = null;  
    // Configuration 对象  
    Configuration targetConfiguration;  
    Optional var10000;  
    if (this.configuration != null) { // 判断是否存在 configuration 对象，如果存在  
说明已经解析过了  
        targetConfiguration = this.configuration;  
        // 覆盖属性  
        if (targetConfiguration.getVariables() == null) {  
            targetConfiguration.setVariables(this.configurationProperties);  
        } else if (this.configurationProperties != null) {  
  
targetConfiguration.getVariables().putAll(this.configurationProperties);  
    }  
    // 如果 configuration 对象不存在，但是存在 configLocation 属性，就根据 mybatis-  
config.xml 的文件路径来构建 xmlConfigBuilder 对象  
    } else if (this.configLocation != null) {  
        xmlConfigBuilder = new  
        XMLConfigBuilder(this.configLocation.getInputStream(), (String)null,  
this.configurationProperties);  
        targetConfiguration = xmlConfigBuilder.getConfiguration();  
    } else {  
        // 属性 'configuration' 或 'configLocation' 未指定，使用默认 MyBatis 配置  
        LOGGER.debug(() -> {  
            return "Property 'configuration' or 'configLocation' not  
specified, using default MyBatis Configuration";  
        });  
        targetConfiguration = new Configuration();  
        var10000 = Optional.ofNullable(this.configurationProperties);  
        Objects.requireNonNull(targetConfiguration);  
        var10000.ifPresent(targetConfiguration::setVariables);  
    }  
    // 设置 Configuration 中的属性 即我们可以在 MyBatis 和 Spring 的整合文件中来设置  
MyBatis 的全局配置文件中的设置  
    var10000 = Optional.ofNullable(this.objectFactory);  
    Objects.requireNonNull(targetConfiguration);  
    var10000.ifPresent(targetConfiguration::setObjectFactory);  
}
```

```
var10000 = Optional.ofNullable(this.objectWrapperFactory);
Objects.requireNonNull(targetConfiguration);
var10000.ifPresent(targetConfiguration::setObjectWrapperFactory);
var10000 = Optional.ofNullable(this.vfs);
Objects.requireNonNull(targetConfiguration);
var10000.ifPresent(targetConfiguration::setVfsImpl);
Stream var24;
if (StringUtils.hasLength(this.typeAliasesPackage)) {
    var24 = this.scanClasses(this.typeAliasesPackage,
this.typeAliasesSuperType).stream().filter((clazz) -> {
        return !clazz.isAnonymousClass();
    }).filter((clazz) -> {
        return !clazz.isInterface();
    }).filter((clazz) -> {
        return !clazz.isMemberClass();
    });
    TypeAliasRegistry var10001 =
targetConfiguration.getTypeAliasRegistry();
    Objects.requireNonNull(var10001);
    var24.forEach(var10001::registerAlias);
}

if (!ObjectUtils.isEmpty(this.typeAliases)) {
    Stream.of(this.typeAliases).forEach((typeAlias) -> {

        targetConfiguration.getTypeAliasRegistry().registerAlias(typeAlias);
        LOGGER.debug(() -> {
            return "Registered type alias: '" + typeAlias + "'";
        });
    });
}

if (!ObjectUtils.isEmpty(this.plugins)) {
    Stream.of(this.plugins).forEach((plugin) -> {
        targetConfiguration.addInterceptor(plugin);
        LOGGER.debug(() -> {
            return "Registered plugin: '" + plugin + "'";
        });
    });
}

if (StringUtils.hasLength(this.typeHandlersPackage)) {
    var24 = this.scanClasses(this.typeHandlersPackage,
TypeHandler.class).stream().filter((clazz) -> {
        return !clazz.isAnonymousClass();
    }).filter((clazz) -> {
        return !clazz.isInterface();
    }).filter((clazz) -> {
        return !Modifier.isAbstract(clazz.getModifiers());
    });
    TypeHandlerRegistry var25 =
targetConfiguration.getTypeHandlerRegistry();
    Objects.requireNonNull(var25);
    var24.forEach(var25::register);
}

if (!ObjectUtils.isEmpty(this.typeHandlers)) {
    Stream.of(this.typeHandlers).forEach((typeHandler) -> {
```

```
targetConfiguration.getTypeHandlerRegistry().register(typeHandler);
    LOGGER.debug(() -> {
        return "Registered type handler: '" + typeHandler + "'";
    });
}

if (!ObjectUtils.isEmpty(this.scriptingLanguageDrivers)) {
    Stream.of(this.scriptingLanguageDrivers).forEach((languageDriver) ->
{
    targetConfiguration.getLanguageRegistry().register(languageDriver);
    LOGGER.debug(() -> {
        return "Registered scripting language driver: '" +
languageDriver + "'";
    });
}

var10000 = Optional.ofNullable(this.defaultScriptingLanguageDriver);
Objects.requireNonNull(targetConfiguration);
var10000.ifPresent(targetConfiguration::setDefaultScriptingLanguage);
if (this.databaseIdProvider != null) {
    try {

targetConfiguration.setDatabaseId(this.databaseIdProvider.getDatabaseId(this.dataSource));
    } catch (SQLException var23) {
        throw new NestedIOException("Failed getting a databaseId",
var23);
    }
}

var10000 = Optional.ofNullable(this.cache);
Objects.requireNonNull(targetConfiguration);
var10000.ifPresent(targetConfiguration::addCache); // 如果cache不为空就把
cache 添加到 configuration对象中
if (xmlConfigBuilder != null) {
    try {
        xmlConfigBuilder.parse(); // 解析全局配置文件
        LOGGER.debug(() -> {
            return "Parsed configuration file: '" + this.configLocation
+ "'";
        });
    } catch (Exception var21) {
        throw new NestedIOException("Failed to parse config resource: "
+ this.configLocation, var21);
    } finally {
        ErrorContext.instance().reset();
    }
}

targetConfiguration.setEnvironment(new Environment(this.environment,
(TransactionFactory)(this.transactionFactory == null ? new
SpringManagedTransactionFactory() : this.transactionFactory), this.dataSource));
if (this.mapperLocations != null) {
    if (this.mapperLocations.length == 0) {
```

```

        LOGGER.warn(() -> {
            return "Property 'mapperLocations' was specified but
matching resources are not found.";
        });
    } else {
        Resource[] var3 = this.mapperLocations;
        int var4 = var3.length;

        for(int var5 = 0; var5 < var4; ++var5) {
            Resource mapperLocation = var3[var5];
            if (mapperLocation != null) {
                try {
                    //创建了一个用来解析Mapper.xml的XMLMapperBuilder，调用了它的parse()方法。这个步骤我们之前了解过了，
                    //主要做了两件事情，一个是把增删改查标签注册成
                    MappedStatement对象。
                    // 第二个是把接口和对应的MapperProxyFactory工厂类注册到
                    MapperRegistry中
                    XMLMapperBuilder xmlMapperBuilder = new
                    XMLMapperBuilder(mapperLocation.getInputStream(), targetConfiguration,
                    mapperLocation.toString(), targetConfiguration.getSqlFragments());
                    xmlMapperBuilder.parse();
                } catch (Exception var19) {
                    throw new NestedIOException("Failed to parse mapping
resource: '" + mapperLocation + "'", var19);
                } finally {
                    ErrorContext.instance().reset();
                }
            }

            LOGGER.debug(() -> {
                return "Parsed mapper file: '" + mapperLocation +
"";
            });
        }
    } else {
        LOGGER.debug(() -> {
            return "Property 'mapperLocations' was not specified.";
        });
    }
}

// 最后调用sqlSessionFactoryBuilder.build()返回了一个
DefaultsqlSessionFactory。
return this.sqlSessionFactoryBuilder.build(targetConfiguration);
}

```

在afterPropertiesSet方法中完成了SqlSessionFactory对象的创建，已经相关配置文件和映射文件的解析操作。

方法小结一下：通过定义一个实现了InitializingBean接口的SqlSessionFactoryBean类，里面有一个afterPropertiesSet()方法会在bean的属性值设置完的时候被调用。Spring在启动初始化这个Bean的时候，完成了解析和工厂类的创建工作。

2.1.2 getObject

另外SqlSessionFactoryBean实现了FactoryBean接口。

FactoryBean的作用是让用户可以自定义实例化Bean的逻辑。如果从BeanFactory中根据Bean的ID获取一个Bean，它获取的其实是FactoryBean的getObject()返回的对象。

也就是说，我们获取SqlSessionFactoryBean的时候，就会调用它的getObject()方法。

```
public SqlSessionFactory getObject() throws Exception {  
    if (this.sqlSessionFactory == null) {  
        this.afterPropertiesSet();  
    }  
  
    return this.sqlSessionFactory;  
}
```

getObject方法中的逻辑就非常简单，返回SqlSessionFactory对象，如果SqlSessionFactory对象为空的话就又调用一次afterPropertiesSet来解析和创建一次。

2.1.3 onApplicationEvent

实现ApplicationListener接口让SqlSessionFactoryBean有能力监控应用发出的一些事件通知。比如这里监听了ContextRefreshedEvent（上下文刷新事件），会在Spring容器加载完之后执行。这里做的事情是检查ms是否加载完毕。

```
public void onApplicationEvent(ApplicationEvent event) {  
    if (this.failFast && event instanceof ContextRefreshedEvent) {  
        this.sqlSessionFactory.getConfiguration().getMappedStatementNames();  
    }  
}
```

2.2 SqlSession

2.2.1 DefaultSqlSession的问题

在前面介绍MyBatis的使用的时候，通过SqlSessionFactory的open方法获取的是DefaultSqlSession，但是在Spring中我们不能直接使用DefaultSqlSession，因为DefaultSqlSession是线程不安全的。所以直接使用会存在数据安全问题，针对这个问题的，在整合的MyBatis-Spring的插件包中给我们提供了一个对应的工具SqlSessionTemplate。

```
import ...  
/*  
 * The default implementation for {@link SqlSession}.  
 * Note that this class is not Thread-Safe.  
 */  
* @author Clinton Begin  
*/  
public class DefaultSqlSession implements SqlSession {  
  
    private final Configuration configuration;  
    private final Executor executor;  
  
    private final boolean autoCommit;  
    private boolean dirty;  
    private List<Cursor<?>> cursorList;  
  
    public DefaultSqlSession(Configuration configuration, Executor executor)  
    {  
        this.configuration = configuration;  
        this.executor = executor;  
        this.dirty = false;  
        this.autoCommit = autoCommit;  
    }  
  
    public DefaultSqlSession(configuration, Executor executor)  
    {  
        this.configuration = configuration;  
        this.executor = executor;  
        this.dirty = false;  
        this.autoCommit = autoCommit;  
    }  
  
    public void close()  
    {  
        configuration.close();  
        executor.close();  
        cursorList.clear();  
    }  
}
```

<https://mybatis.org/mybatis-3/zh/getting-started.html>

作用域 (Scope) 和生命周期

理解我们之前讨论过的不同作用域和生命周期类别是至关重要的，因为错误的使用会导致非常严重的并发问题。

提示 对象生命周期和依赖注入框架

依赖注入框架可以创建线程安全的、基于事务的 SqlSession 和映射器，并将它们直接注入到你的 bean 中，因此可以直接忽略它们的生命周期。如果对如何通过依赖注入框架使用 MyBatis 感兴趣，可以研究一下 MyBatis-Spring 或 MyBatis-Guice 两个子项目。

SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是局部方法变量）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但最好还是不要一直保留着它，以保证所有的 XML 解析资源可以被释放给更重要的事情。

SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏习惯”。因此 SqlSessionFactory 的最佳作用域是应用域。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将 SqlSession 实例的引用放在任何类型的托管作用域中，比如 Servlet 框架中的 HttpSession。如果你现在正在使用一种 Web 框架，考虑将 SqlSession 放在一个和 HTTP 请求相似的作用域中。换句话说，每次收到 HTTP 请求，就可以打开一个 SqlSession，返回一个响应后，就关闭它。这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 finally 块中。下面的示例就是一个确保 SqlSession 关闭的标准模式：

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    // 你的应用逻辑代码  
}
```

在所有代码中都遵循这种使用模式，可以保证所有数据库资源都能被正确地关闭。

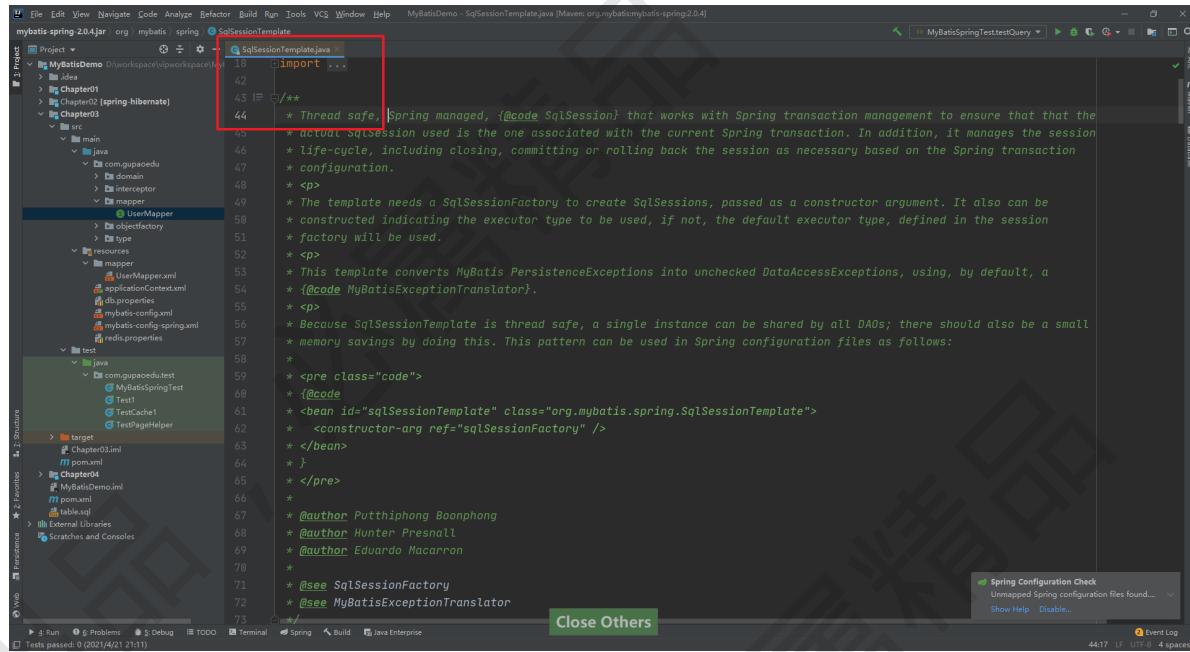
也就是在我们使用SqlSession的时候都需要使用try catch 块来处理

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    // 你的应用逻辑代码  
}  
// 或者  
SqlSession session = null;  
try {  
    session = sqlSessionFactory.openSession();  
    // 你的应用逻辑代码  
}finally{  
    session.close();  
}
```

在整合Spring中通过提供的SqlSessionTemplate来简化了操作，提供了安全处理。

2.2.2 SqlSessionTemplate

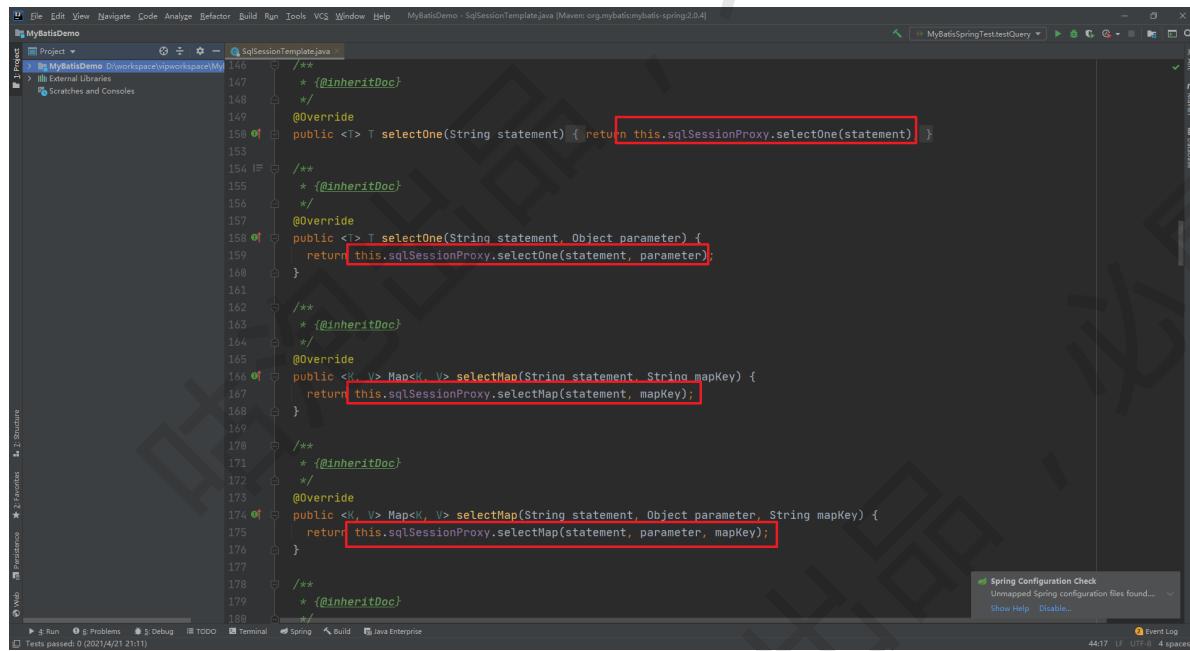
在mybatis-spring的包中，提供了一个线程安全的SqlSession的包装类，用来替代SqlSession，这个类就是SqlSessionTemplate。因为它是线程安全的，所以可以在所有的DAO层共享一个实例（默认是单例的）。



The screenshot shows the IntelliJ IDEA interface with the file `SqlSessionTemplate.java` open. The code is annotated with JavaDoc comments explaining its thread safety and integration with Spring transaction management. The code itself is mostly empty, delegating most operations to a proxy object.

```
1.8 import ...
42 /**
43 * Thread safe, spring managed, {@code SqlSession} that works with Spring transaction management to ensure that the
44 * actual sqlSession used is the one associated with the current Spring transaction. In addition, it manages the session
45 * life-cycle, including closing, committing or rolling back the session as necessary based on the Spring transaction
46 * configuration.
47 */
48 *
49 * The template needs a SqlSessionFactory to create SqlSessions, passed as a constructor argument. It also can be
50 * constructed indicating the executor type to be used, if not, the default executor type, defined in the session
51 * factory will be used.
52 */
53 *
54 * This template converts MyBatis PersistenceExceptions into unchecked DataAccessExceptions, using, by default, a
55 * {@code MyBatisExceptionTranslator}.
56 */
57 *
58 * Because SqlSessionTemplate is thread safe, a single instance can be shared by all DAOs; there should also be a small
59 * memory savings by doing this. This pattern can be used in Spring configuration files as follows:
60 *
61 * <pre class="code">
62 * <bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
63 *   <constructor-arg ref="sqlSessionFactory" />
64 * </bean>
65 * </pre>
66 *
67 * @author Putthiphong Boonphong
68 * @author Hunter Presnall
69 * @author Eduardo Macarron
70 *
71 * @see SqlSessionFactory
72 * @see MyBatisExceptionTranslator
73 *
```

SqlSessionTemplate虽然跟DefaultSqlSession一样定义了操作数据的selectOne()、selectList()、insert()、update()、delete()等所有方法，但是没有自己的实现，全部调用了一个代理对象的方法。



The screenshot shows the IntelliJ IDEA interface with the file `SqlSessionTemplate.java` open. The code is annotated with JavaDoc comments. Several methods are highlighted with red boxes, showing they delegate to a proxy object (`this.sqlSessionProxy`). These include `selectOne`, `selectOne` with parameters, `selectMap`, and `selectMap` with parameters.

```
146 /**
147 * {@inheritDoc}
148 */
149 @Override
150 public <T> T selectOne(String statement) { return this.sqlSessionProxy.selectOne(statement); }
151
152 /**
153 * {@inheritDoc}
154 */
155 @Override
156 public <T> T selectOne(String statement, Object parameter) {
157     return this.sqlSessionProxy.selectOne(statement, parameter);
158 }
159
160 /**
161 * {@inheritDoc}
162 */
163 @Override
164 public <K, V> Map<K, V> selectMap(String statement, String mapKey) {
165     return this.sqlSessionProxy.selectMap(statement, mapKey);
166 }
167
168 /**
169 * {@inheritDoc}
170 */
171 @Override
172 public <K, V> Map<K, V> selectMap(String statement, Object parameter, String mapKey) {
173     return this.sqlSessionProxy.selectMap(statement, parameter, mapKey);
174 }
175
176 /**
177 * {@inheritDoc}
178 */
179
```

那么SqlSessionProxy是怎么来的呢？在SqlSessionTemplate的构造方法中有答案

```
public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType
executorType,
PersistenceExceptionTranslator exceptionTranslator) {
    notNull(sqlSessionFactory, "Property 'sqlSessionFactory' is required");
    notNull(executorType, "Property 'executorType' is required");
    this.sqlSessionFactory = sqlSessionFactory;
}
```

```

        this.executorType = executorType;
        this.exceptionTranslator = exceptionTranslator;
        // 创建了一个 SqlSession 接口的代理对象，调用SqlSessionTemplate中的 selectOne()
方法，其实就是调用
        // SqlSessionProxy的 selectOne() 方法，然后执行的是 SqlSessionInterceptor里面的
        invoke方法
        this.sqlSessionProxy = (SqlSession)
newProxyInstance(SqlSessionFactory.class.getClassLoader(),
    new Class[] { SqlSession.class }, new SqlSessionInterceptor());
}

```

通过上面的介绍那么我们应该进入到 SqlSessionInterceptor 的 invoke 方法中。

```

1 本质上是获取的是
DefaultSqlSession对象
2 执行 DefaultSqlSession.selectOne()
3 对应的方法
4 如果MyBatis中的数据库操作出现了问题就抛出统一的异常信息
5 关闭Session，本质上是关闭的 DefaultSqlSession 对象

```

上面的代码虽然看着比较复杂，但是本质上就是下面的操作

```

SqlSession session = null;
try {
    session = sqlSessionFactory.openSession();
    // 你的应用逻辑代码
}finally{
    session.close();
}

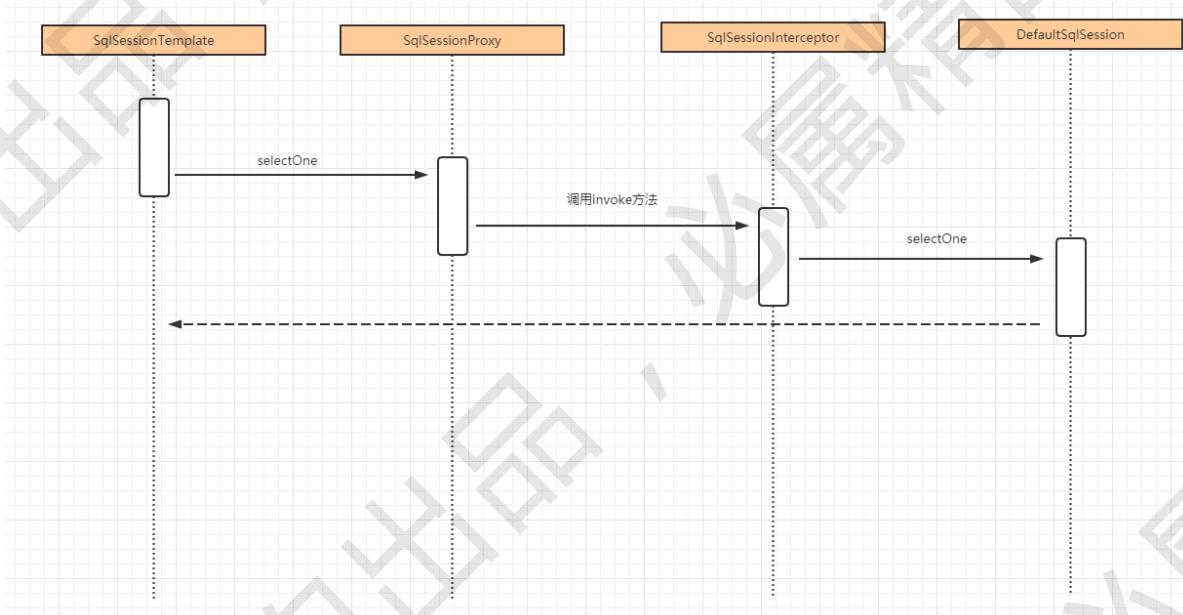
```

getSqlSession方法中的关键代码：

```

    public static SqlSession getSqlSession(SqlSessionFactory sessionFactory, ExecutorType executorType,
                                          PersistenceExceptionTranslator exceptionTranslator) {
        ...
        session = sessionFactory.openSession(executorType);
        ...
    }

```



总结一下：因为DefaultSqlSession自己做不到每次请求调用产生一个新的实例，我们干脆创建一个代理类，也实现SqlSession，提供跟DefaultSqlSession一样的方法，在任何一个方法被调用的时候都先创建一个DefaultSqlSession实例，再调用被代理对象的相应方法。

MyBatis还自带了一个线程安全的SqlSession实现：SqlSessionManager，实现方式一样，如果不集成到Spring要保证线程安全，就用SqlSessionManager。

2.2.3 SqlSessionDaoSupport

通过上面的介绍我们清楚了在Spring项目中我们应该通过SqlSessionTemplate来执行数据库操作，那么我们就应该首先将SqlSessionTemplate添加到IoC容器中，然后我们在Dao通过@Autowired来获取具体步骤参考官网：<http://mybatis.org/spring/zh/sqlsession.html>

```

<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>

@Configuration
public class MyBatisConfig {
    @Bean
    public SqlSessionTemplate sqlSession() throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory());
    }
}

现在,这个bean就可以直接注入到你的DAO bean中了。你需要在你的bean中添加一个SqlSession属性,就像下面这样:

public class UserDaoImpl implements UserDao {
    private SqlSession sqlSession; 2

    public void setSqlSession(SqlSession sqlSession) {
        this.sqlSession = sqlSession;
    }

    public User getUser(String userId) {
        return sqlSession.selectOne("org.mybatis.sample.mapper.UserMapper.getUser", userId);
    }
}

接下来说说,注入 SqlSessionTemplate:

<bean id="userDao" class="org.mybatis.spring.sample.dao.UserDaoImpl">
    <property name="sqlSession" ref="sqlSession" />
</bean>

SqlSessionTemplate 还有一个接收 ExecutorType 参数的构造方法,这允许你使用如下 Spring 配置来批量创建对象,例如批量创建一些SqlSession:
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
    <constructor-arg index="1" value="BATCH" />
</bean>

@Configuration
public class MyBatisConfig {
    ...
    public SqlSessionTemplate sqlSession() throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory(), ExecutorType.BATCH);
    }
}

```

然后我们可以看看SqlSessionDaoSupport中的代码

```

private SqlSessionTemplate sqlSessionTemplate;

public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
    if (this.sqlSessionTemplate == null || sqlSessionFactory != this.sqlSessionTemplate.getSqlSessionFactory()) {
        this.sqlSessionTemplate = createSqlSessionTemplate(sqlSessionFactory);
    }
}

```

如此一来在Dao层我们就只需要继承 SqlSessionDaoSupport 就可以通过getSqlSession方法来直接操作了。

```

public abstract class SqlSessionDaoSupport extends DaoSupport {

    private SqlSessionTemplate sqlSessionTemplate;

    public SqlSession getSqlSession() {
        return this.sqlSessionTemplate;
    }

    // 其他代码省略
}

```

也就是说我们让DAO层（实现类）继承抽象类SqlSessionDaoSupport，就自动拥有了getSqlSession()方法。调用getSqlSession()就能拿到共享的SqlSessionTemplate。

在DAO层执行SQL格式如下：

```
getSqlSession().selectOne(statement, parameter);
getSqlSession().insert(statement);
getSqlSession().update(statement);
getSqlSession().delete(statement);
```

还是不够简洁。为了减少重复的代码，我们通常不会让我们的实现类直接去继承 SqlSessionDaoSupport，而是先创建一个BaseDao继承SqlSessionDaoSupport。在BaseDao里面封装对数据库的操作，包括selectOne()、selectList()、insert()、delete()这些方法，子类就可以直接调用。

```
public class BaseDao extends SqlSessionDaoSupport {
    //使用sqlSessionFactory
    @Autowired
    private SqlSessionFactory sqlSessionFactory;

    @Autowired
    public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
        super.setSqlSessionFactory(sqlSessionFactory);
    }

    public Object selectOne(String statement, Object parameter) {
        return getSqlSession().selectOne(statement, parameter);
    }
    // 后面省略
}
```

然后让我们的DAO层实现类继承BaseDao并且实现我们的Mapper接口。实现类需要加上@Repository的注解。

在实现类的方法里面，我们可以直接调用父类（BaseDao）封装的selectOne()方法，那么它最终会调用 sqlSessionTemplate的selectOne()方法。

```
@Repository
public class EmployeeDaoImpl extends BaseDao implements EmployeeMapper {
    @Override
    public Employee selectByPrimaryKey(Integer empId) {
        Employee emp = (Employee)
this.selectOne("com.gupaoedu.crud.dao.EmployeeMapper.selectByPrimaryKey", empId);
        return emp;
    }
    // 后面省略
}
```

然后在需要使用的地方，比如Service层，注入我们的实现类，调用实现类的方法就行了。我们这里直接在单元测试类DaoSupportTest.java里面注入：

```
@Autowired
EmployeeDaoImpl employeeDao;

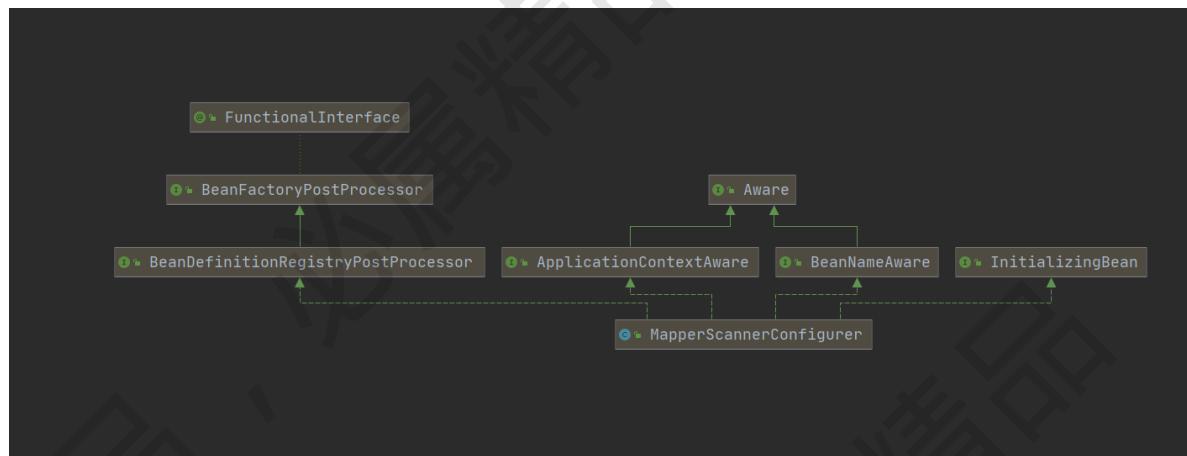
@Test
public void EmployeeDaoSupportTest() {
    System.out.println(employeeDao.selectByPrimaryKey(1));
}
```

最终会调用到DefaultSqlSession的方法。

2.2.4 MapperScannerConfigurer

上面我们介绍了SqlSessionTemplate和SqlSessionDaoSupport，也清楚了他们的作用，但是我们在实际开发的时候，还是能够直接获取到Mapper的代理对象，并没有创建Mapper的实现类，这个到底是怎么实现的呢？这个我们就要注意在整合MyBatis的配置文件中除了SqlSessionFactoryBean以外我们还设置了一个MapperScannerConfigurer，我们来分析下这个类

首先是MapperScannerConfigurer的继承结构



MapperScannerConfigurer实现了BeanDefinitionRegistryPostProcessor接口。BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor的子类，里面有一个postProcessBeanDefinitionRegistry()方法。

实现了这个接口，就可以在Spring创建Bean之前，修改某些Bean在容器中的定义。Spring创建Bean之前会调用这个方法。

```
@Override
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
{
    if (this.processPropertyPlaceHolders) {
        processPropertyPlaceHolders(); // 处理 占位符
    }

    // 创建 ClassPathMapperScanner 对象
    ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
    scanner.setAddToConfig(this.addToConfig);
    scanner.setAnnotationClass(this.annotationClass);
    scanner.setMarkerInterface(this.markerInterface);
    scanner.setSqlSessionFactory(this.sqlSessionFactory);
    scanner.setSqlSessionTemplate(this.sqlSessionTemplate);
    scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);
    scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName);
    scanner.setResourceLoader(this.applicationContext);
    scanner.setBeanNameGenerator(this.nameGenerator);
    scanner.setMapperFactoryBeanClass(this.mapperFactoryBeanClass);
    if (StringUtils.hasText(lazyInitialization)) {
        scanner.setLazyInitialization(Boolean.valueOf(lazyInitialization));
    }

    // 根据上面的配置生成对应的 过滤器
    scanner.registerFilters();
    // 开始扫描basePackage字段中指定的包及其子包
    scanner.scan(
        StringUtils.tokenizeToStringArray(this.basePackage,
        ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS));
}
```

上面代码的核心是 scan方法

```
public int scan(String... basePackages) {
    int beanCountAtScanStart = this.registry.getBeanDefinitionCount();
    this.doScan(basePackages);
    if (this.includeAnnotationConfig) {
        AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
    }

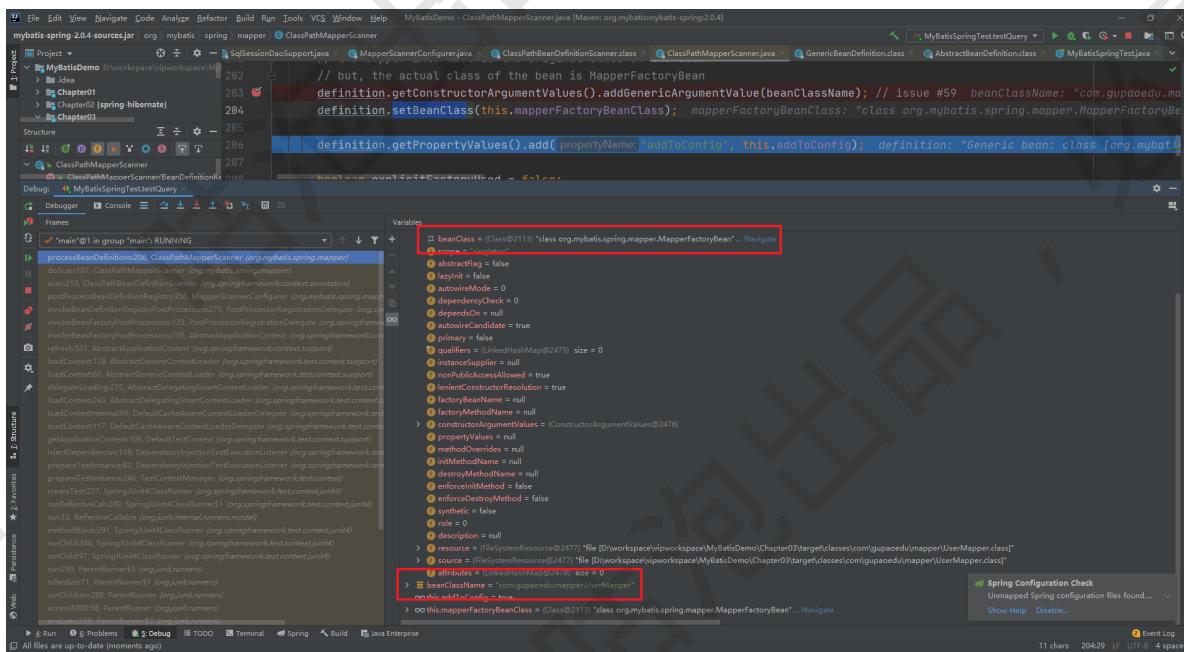
    return this.registry.getBeanDefinitionCount() - beanCountAtScanStart;
}
```

然后会调用子类 ClassPathMapperScanner 中的 doScan方法

```
@Override
public Set<BeanDefinitionHolder> doScan(String... basePackages) {
    // 调用父类中的 doScan方法 扫描所有的接口，把接口全部添加到beanDefinitions中。
    Set<BeanDefinitionHolder> beanDefinitions = super.doScan(basePackages);

    if (beanDefinitions.isEmpty()) {
        LOGGER.warn(() -> "No MyBatis mapper was found in '" +
        Arrays.toString(basePackages)
        + "' package. Please check your configuration.");
    } else {
        // 在注册beanDefinitions的时候，BeanClass被改为MapperFactoryBean
        processBeanDefinitions(beanDefinitions);
    }

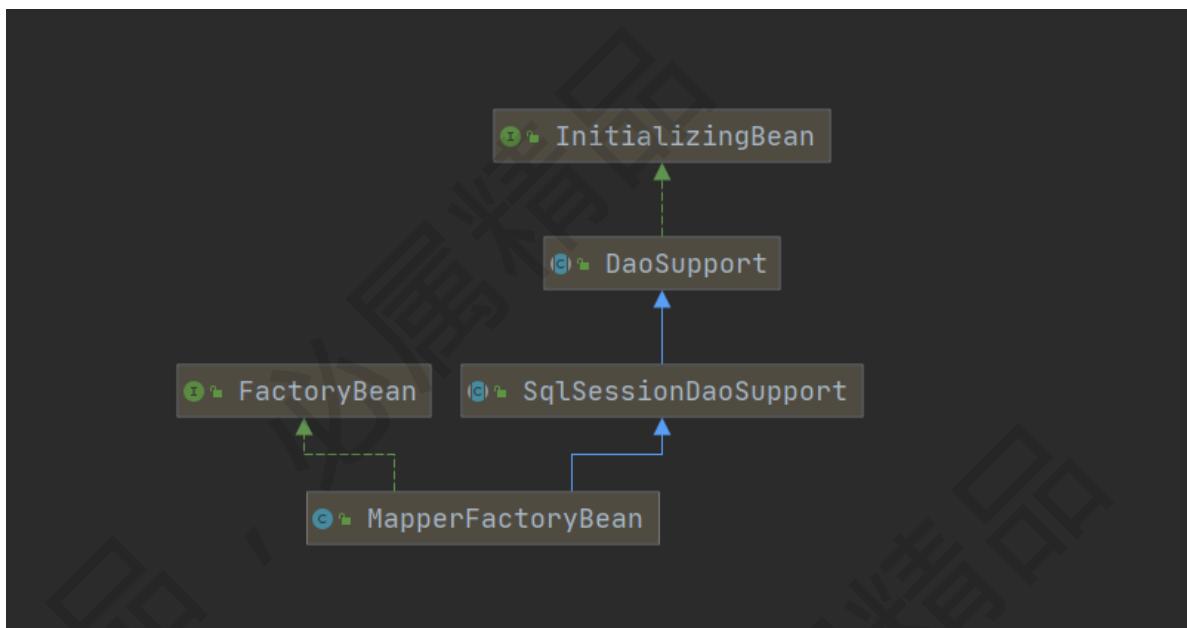
    return beanDefinitions;
}
```



因为一个接口是没法创建实例对象的，这时我们就在创建对象之前将这个接口类型指向了一个具体的普通Java类型，MapperFactoryBean。也就是说，所有的Mapper接口，在容器里面都被注册成一个支持泛型的MapperFactoryBean了。然后在创建这个接口的对象时创建的就是MapperFactoryBean 对象。

2.2.5 MapperFactoryBean

为什么要注册成它呢？那注入使用的时候，也是这个对象，这个对象有什么作用？首先来看看他们的类图结构



从类图中我们可以看到MapperFactoryBean继承了SqlSessionDaoSupport,那么每一个注入Mapper的地方，都可以拿到SqlSessionTemplate对象了。然后我们还发现MapperFactoryBean实现了FactoryBean接口，也就意味着，向容器中注入MapperFactoryBean对象的时候，本质上是把getObject方法的返回对象注入到了容器中，

```
/** * {@inheritDoc} */ @Override public T getObject() throws Exception { // 从这可以看到 本质上 Mapper接口 还是通过DefaultSqlSession.getMapper方法获取了一个 JDBC的代理对象，和我们前面讲解的就关联起来了 return getSqlSession().getMapper(this.mapperInterface); }
```

它并没有直接返回一个MapperFactoryBean。而是调用了SqlSessionTemplate的getMapper()方法。SqlSessionTemplate的本质是一个代理，所以它最终会调用DefaultSqlSession的getMapper()方法。后面的流程我们就不重复了。也就是说，最后返回的还是一个JDK的动态代理对象。

所以最后调用Mapper接口的任何方法，也是执行MapperProxy的invoke()方法，后面的流程就跟编程式的工程里面一模一样了

###

总结一下，Spring是怎么把MyBatis继承进去的？

- 1、提供了SqlSession的替代品SqlSessionTemplate，里面有一个实现了InvocationHandler的内部SqlSessionInterceptor，本质是对SqlSession的代理。
- 2、提供了获取SqlSessionTemplate的抽象类SqlSessionDaoSupport。
- 3、扫描Mapper接口，注册到容器中的是MapperFactoryBean，它继承了SqlSessionDaoSupport，可以获得SqlSessionTemplate。

4、把Mapper注入使用的时候，调用的是getObject()方法，它实际上是调用了SqlSessionTemplate的getMapper()方法，注入了一个JDK动态代理对象。

5、执行Mapper接口的任意方法，会走到触发管理类MapperProxy，进入SQL处理流程。

核心对象：

对象	生命周期
SqlSessionTemplate	Spring中SqlSession的替代品，是线程安全的
SqlSessionDaoSupport	用于获取SqlSessionTemplate
SqlSessionInterceptor (内部类)	代理对象，用来代理DefaultSqlSession，在SqlSessionTemplate中使用
MapperFactoryBean	代理对象，继承了SqlSessionDaoSupport用来获取SqlSessionTemplate
SqlSessionHolder	控制SqlSession和事务

3.设计模式总结

设计模式	类
工厂模式	SqlSessionFactory、ObjectFactory、MapperProxyFactory
建造者模式	XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder
单例模式	SqlSessionFactory、Configuration、ErrorContext
代理模式	绑定：MapperProxy 延迟加载：ProxyFactory 插件：PluginSpring 集成MyBatis：SqlSessionTemplate的内部SqlSessionInterceptorMyBatis 自带连接池：PooledConnection 日志打印：ConnectionLogger、StatementLogger
适配器模式	Log，对于Log4j、JDK logging这些没有直接实现slf4j接口的日志组件，需要适配器
模板方法	BaseExecutor、SimpleExecutor、BatchExecutor、ReuseExecutor
装饰器模式	LoggingCache、LruCache对PerpetualCacheCachingExecutor对其他Executor
责任链模式	Interceptor、InterceptorChain

